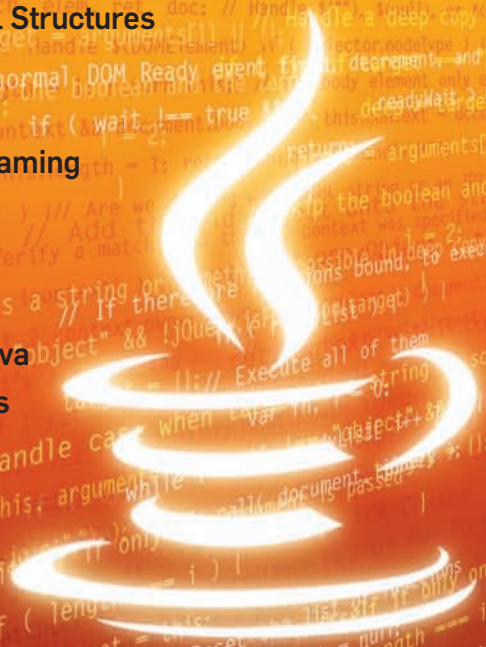
VOLUME 06 | ISSUE 01

digit FastTrack

YOUR HANDY GUIDE TO EVERYDAY TECHNOLOGY

to JAVA

- Java Applications
- Operators and Control Structures
- Methods
- Objects and Classes
- Object Oriented Programing
- Arrays and Strings
- Files and Streams
- Multithreading
- Graphics and 2D in Java
- Java GUI Components



A 9 9 Media Publication

Fast Track

to

Java

Credits

The People Behind This Book

Editorial

Editor	Robert Sovereign-Smith
Head-Copy Desk	Nash David
Writer	Hanu Prateek, Rahil Banthia

Design and Layout

Lead Designer	Chander Dange
Senior Designer	Baiju NV
Cover Design	Prasanth T R

January 2011

Free with Digit. Not to be sold separately. If you have paid separately for this book, please email the editor at editor@thinkdigit.com along with details of location of purchase, for appropriate action.

Contents

Introduction

It all started with an angry programmer in 1990. A Sun Microsystems Software Engineer Patrick Naughton, aged 25, was disgusted with his job at Sun. He had the impossible job of making different software APIs from dozens of languages, platform OS's and vendors all to work as one. It was impossible. Naughton announced to CEO Scott McNealy that he was quitting Sun. Pat was going to join NeXT, because Sun was a mess. McNealy asked Pat to write a memo outlining his gripes. The CEO asked Pat to suggest a solution, stating "Think As if you were God." This was the initiation of the project leading upto Java.

By January 1995, the Internet had taken off. Bill Joy, one of Sun's founders, realised the needs of the Web reliability, platform independence, and security exactly matched the characteristics of Oak, which had just been renamed Java. Although Java had not yet been officially released in 1995, it was spreading like wildfire among developers. Then, something very lucky happened. Two months later, at the SunWorld conference, Marc Andreessen stepped on stage and announced that "Java is real, and it will be incorporated into Netscape Navigator 2.0."

Times have changed. The entity formerly known as Sun Microsystems now belongs to Oracle. What hasn't changed is the significance of Java in rich multimedia applications and the way we consume them. In this Fast Track to Java, we have tried to cover the most important topics.

The Java Platform is unique, because it can work without modification on any platform, on any operating system, if that platform has a "Java Virtual Machine." In a typical C program, the source code is compiled into a native machine language module that consists of 1's and 0's. The machine language is specifically tailored to one OS, be it Windows, Mac or Linux. Therefore, it is impossible for one object module to be portable between platforms. In contrast to conventional programming languages, a Java program

is not compiled into native machine language. Instead, Java makes bytecode. Bytecode is the result of a "compile", but the compile results in an intermediate form that stops short of native machine-specific code. Instead of making a machine language native code for each particular OS, Java makes a single, universal bytecode module that feeds into any Java Virtual Machine (JVM). Each OS has its own different implementation of the Java Virtual Machine. The JVM sets up its own world within your RAM. It creates an internal software-only sub-computer within the OS. The bytecode talks to the JVM, and the JVM talks to the Operating System. So bascally "Write Once, Run Anywhere". The Virtual Machine interprets the bytecode one instruction at a time, and translates it into native machine code. You compile your program once into bytecode, but it is interpreted anew every time it runs.

In chapter 1, we cover the basic principles behind Java applications. Here we look at the way code is written to build applications.

In chapter 2, we discuss the various operators used in Java. Chapters 3, 4 and 5 discuss terms such as Methods, Objects and classes, and object oriented programming.

Data structure concepts such as Arrays and strings are discussed in chapter 6. Similarly, Files and streams are covered in chapter 7. Multithreading, a concept that enables multiple simultaneous lines of execution, is covered in chapter 8.

Chapter 9 and 10 is about graphics and GUI.

1 Java applications

Let us start our introduction to Java with a basic application. It looks like this:

```
public class Welcome1
     {
            public static void main( String args[] )
                  {
                        System.out.println(   "Fast-
Track to Java!" );
                  }
// end of main()
     }
// end of class Welcome1
```

This makes no sense, right? We will provide a basic overview and then get started with actually chucking out Java code. The first line of the program is the class definition for Welcome1. It is a good practice to begin the class name with an upper-case letter. This program is saved in a file called Welcome1.java. The rest of the code is within this class. The compiler compiles the same file and generates a Welcome1.class which is eventually run on the Java Virtual Machine (JVM). `Welcome1` here is an identifier, which is a user-defined word, consisting of letters, digits, _ (underscore), $ (a dollar sign). Note: An identifier cannot begin with a digit.

1.1 Building an application

We will start by building an application that adds two numbers. Whenever you build any Java Application, there is a set template for design that you automatically follow.

```
import javax.swing.JOptionPane;
```

The compiler imports classes you will be using in your Application.

```
import javax.swing.JOptionPane;
public class Addition
{
} // end of class Addition
```

Define your class name, and right away place the opening and closing brackets--with the comment.

```
import javax.swing.JOptionPane;
```

```
public class Addition
{
            public static void main( String args[] )
                  {
                        System.exit( 0 );
                  } // end of main()
} // end of class Addition
```

Add the main method, and `System.exit( 0 )`.

Next, we actually start with the program logic for the intended application.

```
import javax.swing.JOptionPane;
public class Addition
{
public static void main( String args[] )
{
String firstNumber, secondNumber;
```

These two are "String" references. That means they have the potential to point to objects of type `String`. However, at this point they are empty references.

```
import javax.swing.JOptionPane;
public class Addition
{
public static void main( String args[] )
{
String firstNumber, secondNumber;
Int number1, number2, sum;
System.exit( 0 );
} // end of main()
} // end of class Addition
```

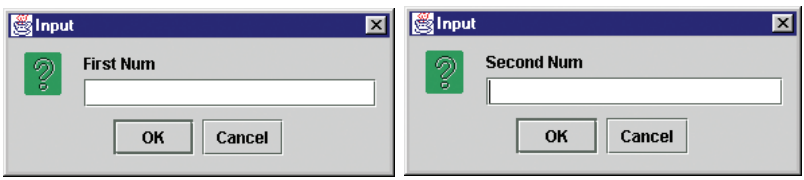
Now, we add three integer variables. They are not objects, and hold three integers (without any methods or classes). `number1`, `number2` and `number3` are called primitive variables.

```
import javax.swing.JOptionPane;
public class Addition
{
public static void main( String args[] )
{
String firstNumber, secondNumber;
```

```
Int number1, number2, sum;
firstNumber = JOptionPane.showInputDialog( "First Num" );
secondNumber = JOptionPane.showInputDialog( "Second Num" );
```

Method `showInputDialog()` receives a String argument, and returns a String result. `JOptionPane` creates InputDialog boxes.

But, since the numbers are still Strings, we can't add them yet. We need some object that has a method capable of taking a String argument and returning an integer.



JOptionPane creates InputDialog boxes

```
import javax.swing.JOptionPane;
public class Addition
{
public static void main( String args[] )
{
String firstNumber, secondNumber;
Int number1, number2, sum;
firstNumber = JOptionPane.showInputDialog( "First Num" );
secondNumber = JOptionPane.showInputDialog( "Second Num" );
number1 = Integer.parseInt( firstNumber );
number2 = Integer.parseInt( secondNumber );
sum = number1 + number2;
```

Integer is a class. Its method `parseInt()` takes a String argument and returns an int.

And finally, the code looks like:

```
import javax.swing.JOptionPane;
public class Addition
{
```

```
        public static void main( String args[] )
        {
                String firstNumber,
                       secondNumber;
                int    number1,
                       number2,
                       sum;
                firstNumber  =  JOptionPane.showInputDialog(
"First Num" );
                secondNumber = JOptionPane.showInputDialog(
"Second Num" );
                number1 = Integer.parseInt( firstNumber);

                number2 = Integer.parseInt( secondNumber);
                sum = number1 + number2;

                JOptionPane.showMessageDialog(  null,  "The
Sum is: " + sum,
                                "Results",  JOPtionPane.PLAIN_
MESSAGE );

                System.exit( 0 );
        } // end of main()
  } // end of class Addition
```

The method `show MessageDialog` of class `JOptionPane` takes four arguments:

i) `null --` this will be explained in a later chapter

ii) `"The Sum is:" + sum --` this converts the int sum into a String and concatenates it with the String "The Sum is:"

iii) `"Results"` is the message displayed in the title bar.

iv) `JOptionPane.PLAIN_MESSAGE` defines the icon.

For the icons, you have five alternate constants to choose from:

(1) `JOptionPane.PLAIN_MESSAGE`

(2) `JOptionPane.ERROR_MESSAGE`

(3) `JOptionPane.INFORMATION_MESSAGE`

(4) `JOptionPane.WARNING_MESSAGE`

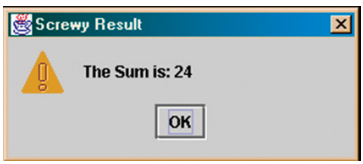
(5) `JOptionPane.QUESTION_MESSAGE`

In Java, Constants are always all upper case, with words separated by

underscores.

Note: We concatenated a String with an int: "The Sum is " + sum. The sequence is as follows: first the sum is converted from an integer to a string, which is then concatenated with the String "The Sum is: ". A common problem lot of people make is writing the code in the following way:

```
int number1 = 2;
int number2 = 4;
JOptionPane.showMessageDialog( null,
            "The Sum is: " + number1 + number2,
            "Screwy Result", JOptionPane.WARNING_MESSAGE
);
```



Error displayed by the code above

1.2 Data type

A variable called number1 actually refers to a place in memory where the value of the variable is stored. Every variable in Java has the following attributes: `name`, `type`, `size`, and a `value`.

1.2.1 Name

Variable names must begin with a letter. After that they can contain digits, the `$` symbol and underscores. Java uses Unicode for its characters, so any "letter" that is valid for a word in any world language is valid for a name in Java.

1.2.2 Type

The "type" appears before the identifier name. It can be one of the "primitive data types" or it can be any previously defined class.

```
int num1;
```

This is a declaration. At this point, the name `num1` refers to a location `{a pointer}` in the computer's RAM where this variable is stored. When an int is declared, four bytes are set aside. Still, nothing is stored in it yet.

1.2.3 Size

When we assign a type `[int, String]` to a variable, we are not only declaring a memory location. We also decide how big of a number or character is able to be stored in that variable.

1.2.4 Value

Finally, `value` is what we want the variable to store.

1.3 Primitive data types

Java is a Strongly-typed language. That means, every variable must be declared as a type. In Java, there are 8 primitive types:

- 6 of those refer to numbers
--4 for integers types,
--2 for floating-point types,
- 1 is the character type char, used for characters in Unicode encoding, and
- 1 is a boolean type for true or false values.

1.3.1 Int

In contrast to C/C++, an int will always--no matter which operating system--take 4 bytes of storage space. Because those 4 bytes are set in stone, you can be sure that every JVM that runs your program will be able to store the same size numbers.

int is the most commonly used number size.

Range: -2,147,483,648 to 2,147,483,647 (over two billion)

1.3.2 Short

In Java, a short is defined as 2 bytes, no matter which operating system is used. You would only use this for special situations, such as when speed is really crucial. For VB programmers, a short is what you've come to think of as an int .

Range: -32,768 to 32,767

1.3.3 long

A long is defined as 8 bytes, no matter which operating system is used.

Range: -9,223,372,036,854,775,808L to 9,223,372,036,854,775,807L

Please notice the upper-case L suffix is appended to any long. This is required.

Hexadecimal numbers have a prefix: 0x

0x1CFE.

1.3.4 byte

A byte is defined as 1 byte, no matter which operating system is used.

Range: -128 to 127

Again, like a short, a byte is only used under rare circumstances.

1.3.5 float

A float is defined as 4 bytes, no matter which operating system is used.

Range: approximately +/- 3.40282347E+38F (6-7 significant decimal digits) because there are so few decimal places available, float is not used all that often.

1.3.6 double

A double is defined as 8 bytes, no matter which operating system is used.

Range: approximately +/- 1.79769313486231570E+308 (15 significant decimal digits). This is the most common choice for any decimal. double is the default, not float, therefore, no special character is appended.

1.3.7 char

A char is defined as 2 bytes, irrespective of the operating system used. A char type always refers to a character in the Unicode encoding scheme. Approximately 65,536 characters can be represented. Single quotes denote a char constant. `'H'` is a char constant. `"H"` is a String that contains a single character – it is not a char. This is a syntax error, and the compiler will give you an error.

1.3.8 boolean

A boolean type has only two values. In contrast to C/C++, in Java, 0 and 1 cannot stand in for true or false. A boolean type must be assigned the value of the constants true or false.

1.4. Java math operators

- Addition +
- Subtraction -
- Multiplication *
- Division /
- Modulus %

All are binary operators, i.e., they work with two numbers. They are executed according to the rules for operator precedence. (There is no operator for exponentiation in Java). Let us see some of the commonly overlooked things regarding Math operators in Java.

1.4.1 Multiplication *

Consider the following code:

```
int x = 2;
double y = 3.889, sum = 0.000;
sum = y * x;
```

The integer will be temporarily converted to a double and two doubles will be multiplied. Afterwards, the original integer remains unchanged.

Rules for temporary conversions:

First priority: If either of the operands is of type double, then the other operand is converted to double for the calculation.

Second priority: If either of the operands is of type float, then the other operand is converted to float for the calculation.

Third Priority: If any of the operands is of type long, then the other operand is converted to long for the calculation.

Note: these conversions are automatic, because none of them result in loss of accuracy.

Static casts

When you convert a double to float, then information is inevitably lost. You can accomplish this using a cast:

```
int x = 2, sum = 0;
double y = 3.889;
sum = (int)y * x;
{ sum is now equal to 6 }
```

Here, a value of just 3 will be used for y. If you want to round y, you a method from class Math:

```
sum = (int)Math.round(y) * x;
```

1.4.2 Division /

Division can lead to unexpected results. For example, if both operands are integers, then the result of the division is also an integer. Any fractional part of the division is discarded. As a result, 17/3 = 5.

1.4.3 Modulus %

In contrast to the division operator, modulus returns the remainder of any division. The modulus operator can only be used when both operands are integers. Thus, 17 % 3 = 2.

1.5 Comparison operators

These are used for selection structures:

- equality ==
- not equal !=
- greater than >
- less than <
- greater than or equal >=
- less than or equal <=

The equality operator is a common source of mistakes. Note that two equal signs are always used. The single equal sign `[ = ]` is only used for assignment, that is, assigning the value on the right to the variable on the left.

```
num1 = 33;
```

When you make a compound symbol using the equal sign, the equal sign should always be placed on the right.

2 Operators and Control Structures

2.1 Assignment operators

The following code assigns the value 15 to the variable x.

```
int x;
x = 15;
```

In Java, a single equals sign is the assignment operator. Consider we append the statement with:

```
x = x + 5;
```

The new value of x will now be 20. First, the addition on the right is done. Then, the result is assigned to the the variable on the left. In Java, you can also use the following shortcut to arrive at the same value:

```
int x = 15;
x = x + 5;
```

2.2 Increment/Decrement operators

There is one addition statement in Java that is so common, it gets its own operator. It means, simply, "Add/Subtract one to the variable." And you can do pre-increment/decrement, or a post-increment/decrement.

```
Pre
++x; x++;
Post
--x; x--;
```

If each of these statements is on a line by itself, there is no difference in the effect of doing a pre- or post-increment. However, if the variable that is having the pre- or post-increment applied to it is used within another statement, your choice of pre- or post-increment can alter your results which will be made clear using this example.

```
// PrePostIncrement.java
public class PrePostIncrement
{
public static void main( String args[] )
{
int x=0;
System.out.println( " Baseline, x = " + x );
System.out.println( "\n Pre-increment = ++x = " + ++x );
System.out.println( "\n After increment, x = " + x );
```

```
  x = 0;
  System.out.println( "\n 2nd Baseline, x = " + x );
  System.out.println( "\n Post-increment = x++ = " + x++
);
  System.out.println( "\n After increment, x = " + x );
  System.exit( 0 );
  }
  }
  Baseline, x = 0
  Pre-increment = ++x = 1
  After increment, x = 1
  2nd Baseline, x = 0
  Post-increment = x++ = 0
  After increment, x = 1
```

2.3 Logical operators

So far, all of the conditions we have tested were simple. It is possible to construct complex conditions using the Java Logical Operators, which—again—were inherited form C/C++.

```
&&      Logical AND
||      Logical OR
!       Logical NOT
```

2.3.1 Logical AND

```
if( gender == 'F' && age >= 65 )
```

Condition is true only if both halves are true. Java will short-circuit the process—skipping the 2nd half of the expression—if the first half is false.

2.3.2 Logical OR

```
if( gender == 'F' || age >= 65 )
```

Entire condition is true if either half is true.

2.3.3 Logical NOT

```
if( !(age <= 65) )
```

Negates the expression—not many opportunities to use.

2.3.4 Logical Boolean AND

```
if( gender == 'F' & ++age >= 65 )
```

A Logical Boolean AND `[ & ]` works exactly like a Logical AND `[ && ]` with one exception. It will always check both halves of the equation, even if the first is false.

2.3.5 Logical inclusive Boolean OR

```
if( gender == 'F' | age >= 65 )
```

Again, this works just like the Logical OR, but you are guaranteed that both sides of the expression will always be executed. If either half is true, the entire 'if' is true.

2.3.6 Logical exclusive Boolean OR

```
if( gender == 'F' ^ age >= 65 )
```

This Logical exclusive Boolean OR is true only if one side is true and the other false. If both sides are true, the entire expression is false.

2.4 Control structures

2.4.1 The "if" Statement

This control statement is used for decision making in any Java application. The if in Javaexactly mirrors C/C++, and it has three variants:

```
if( expression )
statement;
if( expression )
 statement;
 else
statement;
if( expression )
 statement;
 else if( expression )
statement;
 else
statement;
```

The "expression" must be something that uses the comparison operators and resolves to either true or false. The statement is executed if the expression is true. Only one statement can be made conditional without brackets. If you wish to conditionally execute more than one statement, you use brackets to create a block. If the "expression" is not true, the else branch executes.

You can also use the "Ternary" ? : Operator as a shorthand for if / else.

Example: System.out.print(expression ? "True" : "False"). If the expression is True, the command before the colon is executed; and if expression is False, the command after the colon is executed.

2.4.2 while

It's used for repetition in Java. The syntax of this statement is :

```
while( expression )
statement;
```

Alternatively, you can write it as:

```
do
     {
     statements;
     }
     while(expression);
```

Lets see these in real examples.

```
// SelectionTest.java to try the ternary operator
import javax.swing.*;
public class SelectionTest
{
public static void main( String args[] )
{
int b, s;
String big, small, out;
big = JOptionPane.showInputDialog( "Big Number" );
small = JOptionPane.showInputDialog( "Small Number" );
b = Integer.parseInt( big );
s = Integer.parseInt( small );
out = ( b > s ? "Big was larger" : "Small was larger" );
JOptionPane.showMessageDialog( null, out, "Results",
JOptionPane.INFORMATION_MESSAGE);
System.exit( 0 );
}
}
// DoWhileTest.java to try while statement
import javax.swing.*;
public class DoWhileTest
{
```

```
  public static void main( String args[] )
  {
  int b = 2, s = 1;
  String big, small, out = "Big is still Bigger";
  while( b > s )
  {
  JOptionPane.showMessageDialog( null, out, "Results",
JOptionPane.INFORMATION_MESSAGE);
  big = JOptionPane.showInputDialog( "Big Number" );
  small = JOptionPane.showInputDialog( "Small Number" );
  b = Integer.parseInt( big );
  s = Integer.parseInt( small );
  }
  System.exit( 0 );
  }
  }
  // DoUntilTest.java to try the do_while statement
  import javax.swing.*;
  public class DoUntilTest
  {
  public static void main( String args[] )
  {
  int b = 2, s = 1; // preload variables.
  String big, small, out = "Big is still Bigger";
  do
  {
  JOptionPane.showMessageDialog( null, out, "Results",
JOptionPane.INFORMATION_MESSAGE);
  big = JOptionPane.showInputDialog( "Big Number" );
  small = JOptionPane.showInputDialog( "Small Number" );
  b = Integer.parseInt( big );
  s = Integer.parseInt( small );
  }
  while( b > s );
  System.exit( 0 );
  }
  }
```

2.4.3 The for Loop

A common structure called a for loop is specially designed to manage counter-controlled looping.

```
for( int x = 1; x < 10; x++ )
```

An example program implementing the same:

```
// ForCounter.java
import java.awt.Graphics;
import javax.swing.JApplet;
public class ForCounter extends JApplet
{
public void paint( Graphics g )
{
for( int counter=1 ; counter <= 10 ; counter++ )
{
g.drawLine( 10, 10, 250, counter * 10 );
}
}
}
```

When appropriate, `for` is quick and easy. The `for` loop is a do-while. It tests the condition before it executes the loop for the first time.

Note: since the variable int counter was declared within `for`, it vanishes after the `for` is executed. All three sections are optional.

If you omit the condition, Java assumes the statement is true, and you have an infinite loop.

```
for( int x = 1;; x++ )
```

You can omit the initialization if you have initialized the control variable someplace else.

```
int x = 1;
for(; x < 10; x++ )
```

You can omit the increment of the variable if you are doing so within the body of the loop.

```
for( int x = 1; x < 10;)
{
other stuff
x++;
}
```

Let's take another example, this time a more comprehensive one to explain the use of the for loop.

```
// Calculate Compound Interest
import javax.swing.JOptionPane;
import java.text.DecimalFormat;
import javax.swing.JTextArea;
public class Interest
{
public static void main( String args[] )
{
double amount, principle = 1000.0, rate = 0.05;
DecimalFormat twoDig = new DecimalFormat( "0.00" );
JTextArea output = new JTextArea( 11, 20 );
output.append( "Year\tAmount on deposit\n" );
for( int year = 1; year <= 10; year++ )
{
amount = principle * Math.pow( 1.0 + rate, year );
output.append( year + "\t" + twoDig.format(amount) +
"\n");
} // end of for
JOptionPane.showMessageDialog( null, output,
"Compound Interest", JOptionPane.INFORMATION_MESSAGE);
System.exit( 0 );
} // end of main()
} // end of class Interest
```

2.4.4 Multiple-Selection structure

Once you start nesting many 'if's, it becomes a nuisance. Java—like C and C++ before it—provides the switch structure, which provides multiple selections. Unfortunately—in contrast to Visual Basic's Select Case and even COBOL's Evaluate—you cannot use any of type of argument in the switch statement other than an integer.

```
int x = 0;
switch( x )
{
case 1:
do stuff;
break;
case 2:
do stuff;
```

```
break;
case 55:
do stuff;
break;
case 102:
case 299:
do stuff okay for both;
break;
default:
if nothing else do this stuff;
break;
}
```

The integer expression x is evaluated. If x contains a 1, then the case 1 branch is performed. Notice the 'break;' statement. This is required. Without it, every line after the match will be executed until it reaches a break;

The expression within the switch(expression) section must evaluate to an integer. Actually, the expression can evaluate to any of these types (all numeric but long):

- byte
- short
- int
- char

but they will be reduced to an integer and that value will be used in the comparison. Also, the expression after each case statement can only be a constant integral expression or any combination of character constants and integer constants that evaluate to a constant integer value.

The default: is optional. If you omit the default choice, then it is possible for none of your choices to find a match and that nothing will be executed. If you omit the break; then the code for every choice after that—except the default!—will be executed.

2.4.5 break; and continue;

Both of these statements alter the flow of control. The break statement can be executed in a:

- while
- do/while
- for
- switch

It causes the immediate exit from the structure. After a break exits the "structure"—whatever that is—execution resumes with the first statement following the structure. If you have nested structures—be they a while, do/while/ for or switch—the break will only exit the innermost nesting. It will not exit you out of all nests. To do that, you need another break. However, there is a variant of the break called a labeled break—but this is similar to a goto and is frowned upon.

The continue statement, when used in a while or do/while or a for, skips the remaining code in the structure and returns up to the condition. If the condition permits it, the next iteration of the loop is permitted to continue. So, the continue is a "temporary break." It is only used in iterative structures, such as the while, do/while and for.

The "Labeled" continue and break statements send execution to the label to continue execution. Note: using the labeled break and continue is bad code. Avoid using them! We will still give you an example just in case its absolutely required.

```
stop:
for(row = 1; row <= 10; row++)
{
for(col=1; col <=5; col++)
{
if( row == 5)
{
break stop; // jump to stop block
}
output += "* ";
}
output += "\n";
}
```

3 Methods

A Method is invoked by a method call. The Syntax is as follows:

```
object.method(arguments);
```

Usually, we start with an instance of an object. The object hands the method some information (arguments) and asks that method to perform a task based on that information. When the the task is done, the method returns information back to the object that called it.

```
return = object.method(arguments);
```

Example:

```
Teacher.asksStudent( help );
```

The OBJECT (Teacher) does not need to know how the method (student) accomplished the request. When we're writing software, one object is independent of another. How any object implements its own methods is hidden from the outside world. That way, once your method works, nobody can mess it up. Hiding the implementation of a method promotes good software engineering. If you can't change how an object performs its methods, then you can't introduce errors. You have to live with the way it works now.

A Method can receive as many arguments as you wish. But it can only return ONE thing. A Method also always Receives A Copy Of Its Arguments. Since a Method Only Receives a duplicate, it can NEVER change the original copy of the parameter it received.

```
import java.awt.Container;
import javax.swing.*;
public class SquareInt extends JApplet
{
public void init()
{
String output = "";
JTextArea oArea = new JTextArea( 10, 20 );
Container c = getContentPane();
c.add( oArea );
int result;
for( int x = 1; x <= 10; x++ )
{
result = square( x );
```

```
output += ”Square of " + x +
" is " + result + "\n";
}
outputArea.setText( output );
}
public int square( int y )
{
return y * y;
}
}
```

Notice this syntax: We are inside method init(), yet we're calling to another method square() without referring to an object. “Methods in a class definition are allowed to invoke all other methods in the same class this way.” Inherited methods can be called this same way.

All variables declared inside a method are local variables. And local variables must be initialized. They live only while the method is being executed and vanish after the method is done.

```
public double sum( double x, double y)
{
double sum = 0;
sum = x + y;
return sum;
}
```

In this example, all three variables are local. The variables x and y declared as parameters in the header, and the variable sum declared in the body of the method.

3.1 Coercion of Arguments

```
public double sum( double x, double y)
{
double sum = O;
sum = x + y;
return sum;
}
```

What would happen if--when we called this method--we passed it integers, rather than the doubles it expects? If it was able to, the compiler would attempt to convert or “coerce” the arguments into the right type. If I passed

it an integer and it expected a double, the conversion would be no problem--converting an integer into a double doesn't lose any information. However, if I passed it a double and it was expecting an integer, the conversion from double to integer WOULD lose information, so the compiler would complain. The compiler would complain unless I used an explicit cast operator to force the argument into the right type.

3.2 Duration of Identifiers

The duration of an identifier is the lifetime of the identifier. Identifiers declared locally in a method are called automatic. Automatic variables exist only while the block they are declared in executes. Static variables exist from the time the class that defines them is loaded into memory until the program terminates.

3.3 Scope Rules

The scope for an identifier is the portion of the program in which the identifier can be referenced. The scopes for an identifier are:

- class
- block
- method

3.4 Method Overloading

Method overloading allows a method name to be re-used. To overload a method--or to create another version of a method that already exists--the argument lists for the methods must differ in:

- number of arguments
- type of arguments
- order of arguments
- The return type of the method is NOT considered.

3.5 Event Delegation Model

Unlike many other languages--such as Visual Basic--when the Java programmer wishes to program responses to an event--such as the enter key being pressed, the event must be entirely programmed.

Consider this analogy to the event model:

- Steve Jobes decides he wants a bodyguard.
- Somebody punches Steve--that's a problem.
- Bill--the bodyguard--notices Steve was punched.

- Bill reacts to the problem.

The same thing with specifics attached:

- A Button decides it wants an ActionListener.
- A Button is pressed--an event happens:
- The ActionListener notices the Button was pressed.
- The Listener reacts to the event.
- Fully described in Java Terms:
- A Button adds an ActionListener.
- A Button is pressed--an event happens:
- The method actionPerformed is executed, receiving an ActionEvent object.
- The Listener reacts to the event.

An event source is an object that can register listener objects and send those listeners event objects. In practice, the event source can send out event objects to all registered listeners when that event occurs. The listener object(s) will then use the details in the event object to decide how to react to the event. A "Listener Object" [an instance of a class] implements a special interface called a "listener interface." When you ask the listener object to pay attention to your event source, that is called registering the listener object with the source object. You do that with the following lines of code:

```
eventSourceObject.addEventListener( eventListenerObject
);
```

Every event handler requires 3 bits of code:

Code that says the class implements a listener interface:

```
public class MyClass implements ActionListener
```

Code that registers a listener on one or more components.

```
someComponent.addActionListener( MyClass );
```

Code that implements the methods in the listener interface.

```
public void actionPerformed(ActionEvent e)
{
...//code that reacts to the action...
}
```

Lets take another Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Craps extends JApplet implements
```

```
ActionListener
  {
  final int WON = 0, LOST = 1, CONTINUE = 2;
  boolean firstRoll = true;
  int sumOfDice = 0;
  int myPoint = 0;
  int gameStatus = CONTINUE;
  // GUI components.
  JLabel die1Label, die2Label, sumLabel, pointLabel;
  JTextField firstDie, secondDie, sum, point;
  JButton roll;
```

First, to respond to events, we must import the event class API:

```
java.awt.event.*;
```

This is the first time we have seen a class that "implements" another class. This is called an interface. You see, although we are directly inheriting from the class JApplet, we are getting some functions through the interface from the class ActionListener.

```
  // Setup graphical user interface components
  public void init()
  {
  Container c = getContentPane();
  c.setLayout(new FlowLayout() );
  die1Label = new JLabel( "Die 1" );
  c.add( die1Label );
  firstDie = new JTextField( 10 );
  firstDie.setEditable( true );
  c.add( firstDie );
  roll = new JButton( "Roll Dice" );
  roll.addActionListener( this );
  c.add( roll );
  } // end of method init()
```

c is a Container object that came from the Content Pane. Here, we are setting the Layout, or how the objects get stacked on the page. There are many different choices for layouts. We will discuss them later.

Earlier, we declared these JLabels and now we're defining them. Then, we see how the Container c is using its method "add" to add the JLabels object die1Label to its content pane. You can see how we have defined a new

JButton called “roll”. Then, we register an Action Listener on the button roll. We want our same JApplet--the one we are inside of now--to listen to for the events. To say that, we use the “this” reference below. It means “this” JApplet will listen for events that happen to the JButton “roll”. This is called “registering an Action Listener”.

So, when an event happens to our button--because of the ActionListener--the following method is performed. It receives an ActionEvent object. In our example, this method just calls another method, play()

```
// call method play when button is pressed.
public void actionPerformed( ActionEvent e )
{
play();
}


// process one roll of the dice.
public void play()
{
if( firstRoll )
{
sumOfDice = rollDice();
switch( sumOfDice )
{
case 7: case 11:
gameStatus = WON;
point.setText( "" );
break;
case 2: case 3: case 12:
gameStatus = LOST;
point.setText( "" );
break;
} // end of switch
} //end of true if block
else
{
sumOfDice = rollDice();
if( sumOfDice == myPoint )
gameStatus = WON;
```

```
else
```

This method decides what to do when the event happens.

3.6 Interfaces

When we said our program:

```
implements ActionListener
```

it was implementing a thing called an interface.

If you want to implement the interface ActionListener, it means that you must define a method called:

```
public void actionPerformed( ActionEvent e )
```

in your class.

3.7 Recursion

For any programming problem, there are usually two alternative ways to solve it:

iteration - Iteration means repeating the same thing a certain number of times until the solution is achieved.

recursion - Recursion means having a method call itself. Or rather, it's better to think that it calls another copy of itself.

If we choose to call a method recursively, it means we are calling it numerous times until we arrive at the solution. Ironically, a recursive method can only actually solve the simplest "base" case. The other, more complex cases wait until the simplest base case is solved and then it works from the inside out until all the cases are solved.

Recursion Example: The Fibonacci Series

This is a Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

It makes the next number in the sequence by adding together the previous two numbers.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class FibonacciTest extends JApplet implements
ActionListener
{
JLabel numLabel, resultLabel;
JTextField num, result;
public void init()
```

```
  {
  Container c = getContentPane();
  c.setLayout( new FlowLayout() );
  numLabel = new JLabel( "Enter and Integer and press
Enter" );
  c.add( numLabel );
  num = new JTextField( 10 );
  // creates a new JTextField with 10 columns
  num.addActionListener( this );
  // makes "this" Applet listen for events.
  c.add( num );
  resultLabel = new JLabel( "Fibonacci value is " );
  c.add( resultLabel );
  result = new JTextField( 15 );
  result.setEditable( false );
  c.add( result );
  } // end of method init()
  public void actionPerformed( ActionEvent e )
  {
  long number,
  fibonacciValue;
  number = Long.parseLong( num.getText() );
  showStatus( "Calculating..." ); // status area of Applet
  fibonacciValue = fibonacci( number );
  showStatus( "Done" );
  result.setText( Long.toString( fibonacciValue ) );
  } // end of method actionPerformed()
  // Recursive definition of method Fibonacci
  public long fibonacci( long n )
  {
  if( n == 0 || n == 1 ) // base case
  {
  return n;
  }
  else
  {
  return fibonacci( n - 1 ) + fibonacci( n - 2 );
  }
```

```
} // end of method fibonacci()
} // end of class FibonacciTest
public long fibonacci( long n )
{
if( n == 0 || n == 1 ) // base case
{
return n;
}
else
{
return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
}
```

This method is called repeatedly until it reaches the base case, then the previous copies of the method--that had been waiting for the else to finish--complete their execution from the inside out.

4 Objects and classes

4.1 Objects

A program is composed of generic objects, with certain standard properties, and certain standard operations the objects can perform. In object oriented programming (OOP), you only care about what the objects expose. You won't be able to determine how someone else's object functions. In fact, the public method (interface) is more important.

In traditional procedural programming, you search for verbs (actions) in the problem definition. In procedural programming, verbs directly suggest procedures and then, lastly, you think of data variables to go with those procedures. In OOP, you put data structures first, and then look at the algorithms that operate on the data. The secret to effective OOP is that each object carries out a small set of related tasks. If an object needs a task done but that task isn't the job of that object then that object asks another object to do the task.

Again, since the first object can't do the task, it asks the second object to carry out the task. In OOP, one object must never directly manipulate the internal data of another object. Rather, all communication is via "messages". A message is another name for a method call.

4.2 Classes

The term class is the blueprint or recipe from which the object is actually made, or "instantiated."

```
MyClass boop;
boop = new MyClass();
```

We are now familiar with this, the first "MyClass boop; " makes a reference called "boop." At this point, the reference called "boop" does not actually point to any existing object. Soon, it will point to an object of type MyClass, but now the object doesn't exist.

```
MyClass boop = new MyClass();
```

When this statement executes, the new keyword executes the default Constructor for MyClass, which actually creates an object in memory and assigns that reference to boop. The handle to that just-created object is given to the MyClass reference boop. Now boop points to the new MyClass object.

State behavior identity

All instances of a class have the same instance variables, with different values inside them. The state or current values for an instance of a class is called the "state" of that class. The current values of those variables define the current situation or state of this instance of the class. For example, a class called HourlyEmployee contains instance variables:

```
first_name
last_name
soc_sec_number
hourly_rate
current_vacation_time
```

All objects that are instances of the same class share the same behaviour. They all have the same methods. We could send the same messages to all instances of the a class and all would understand and respond to the messages.

Our class is: `HourlyEmployee`

All instances of this class have these methods:

```
calculate_pay()
setName()
getName()
setSSN()
getSSN()
getVacationTime()
setVacationTime()
getHourlyRate()
setHourlyRate()
```

Every example, or instantiation, of this class has the same methods (behaviour) available to it.

Let's instantiate HourlyEmployee :

```
HourlyEmployee joseph; // empty reference.
joseph = new HourlyEmployee('Joe' 'Smith', '598-22-7893', '$10.00', '22.25');
```

Now, we've created an instance of the class HourlyEmployee. Our class is: HourlyEmployee. We have instantiated HourlyEmployee and our instance is called joseph. The identity of the instance is `joseph`.

The state of the instance is:

```
first_name = „Joe
last_name =  Smith
soc_sec_number =  598-22-7893
hourly_rate =  $10.00
current_vacation_time =  22.25
```

The behaviour of the instance is:

```
calculate_pay()
setName()
getName()
setSSN()
getSSN()
```

Now, we shall instantiate three objects:

```
HourlyEmployee ravi;
marie = new HourlyEmployee('Ravi', 'J.', '555-24-1516',
'Rs.30.00', '0');
HourlyEmployee rahul;
theodore = new HourlyEmployee('Ted', 'L.', '681-22-
9875', 'Rs.10.00', '22');
HourlyEmployee priya;
david = new HourlyEmployee('Dave', 'D.', '198-99-0098',
'Rs.15.00', '8');
```

Identity is the reference to this instantiation. All three have the exact same behaviour. The state of each instance is defined by its instance variables. The state of an instance can only be changed by going through its methods or behaviour.

4.2.1 Class scope

A class's Instance variables and methods have a thing called "class scope." Within the class (or scope of that class), class member variables are accessible by name. So, inside or outside of any method in that class, those instance variables can be reached from anywhere in the class. If a member variable has been declared public, then it can be accessed outside of the class by simply referencing as follows:

```
ClassName.primitive_variable
ClassName.Object_variable.
```

Another instance of this class has access to the instance variables in any other instance of this class. You can use the instance identifier or the class name if it is declared as a "static" variable.

4.2.2 Cosmic Base Class

In Java, all classes are built on other classes. Ultimately, all classes in Java stem from one central "Cosmic Base Class" called Object. Even if you didn't use the word "extends" in your class definition, you're still always extending Object by default.

When you extend any "Base Class", the new (derived) class has all the properties (instance variables) and methods of its parent, or Base Class. You can choose to modify or keep any method of the parent, or you can create methods that only apply to the child or "inherited" class.

The concept of extending a base class is called "Inheritance." Inheritance is the second fundamental concept of Object-Oriented programming. Encapsulation is the first, Polymorphism is the third.

4.2.3 Relationships between classes

Classes can be related to each other in one of three alternative ways:

```
use
containment ( "has-a" )
inheritance ( "is-a" )
```

When one class sends messages to another class, we say it "uses" the class that receives its messages. When one class lives as an Instance Variable within another class, we say it is "Contained", a "has-a" relationship. When one class inherits from another class, we say it is an "is-a" relationship.

Imagine that we have a class Order. Class Order needs to use the class Account, in order to check for credit status. Generally, if a method of class Order sends a message to an object of class Account, then Order uses Account. In other words, Order uses Account when Order calls methods of Account.

A method of Order: creates, receives or returns objects of class Account. The "Containment" relationship also known as the "Composition" relationship is a special case of the "use" relationship. In a Containment / Composition relationship, at least one method of one class actually contains an object of another class. In the use relationship, it calls methods of another

object. In the containment relationship, it contains another object. In a "has-a" relationship, a class becomes an instance variable for the class we are defining.

Inheritance means specialisation. When we inherit from a class, we wish to keep nearly everything in the base class (Superclass). In inheritance, we seek to elaborate on what we receive from the Superclass. We start with the class Order. Then, we wish to create a Subclass off of Order. Our Subclass is called RushOrder.

Class RushOrder has everything that Order has, but it adds a few instance variables, maybe adds a method or two and overrides a method or two.

The three relationships between classes which form the foundation of Object-Oriented Design are, use, "has-a" and "is-a"

4.2.4 Techniques for using objects

We have spent a lot of time emphasising the difference between a reference and the object to which it refers.

```
JLabel howdy;
howdy = new JLabel( "How Are Ya?" );
```

Here howdy is a reference. We start off by declaring a reference "howdy" to an object of type JLabel. Then, we instantiate the object by calling its constructor with the new keyword, and assign the handle to this instantiation to the reference we declared: "howdy".

```
howdy = new JLabel( "How Are Ya?" );
```

Okay, what happens when we execute the following statement?

```
JLabel hello; // A new reference
hello = howdy;
```

Now, both references point to the exact same object. Any changes made from howdy will be reflected in hello.

4.2.5 Controlling access to methods and variables

`Public` lets clients see the services (methods) the class provides (which means view the interface). The interface is the collective name for all the various methods that are available in the class. Methods should be public.

`Private` is the default setting. It hides implementation details. Private data members (variables) are only accessible through the public interface (Accessor methods) using public methods. Only utility methods should be made private. Utility methods are used only within the class.

`Package` is used when you don't specify that a method or a data variable is either private or public. If your program has only one class definition this change is transparent. If you don't specify either public or private for any feature meaning class, method or variable it can be accessed by all methods in the same package. So, if you have the following field in your class:

```
public class MyClass
{
int mySalary;
…
}
…
```

and your class is stored in java.util.*; then any other method in any class that is also stored in this package can change this variable to anything it wants. What's more, anybody can add their own class to any package. And if you have a method to exploit that variable with package access, you could do anything you wanted! So, if your program uses many classes that are stored in the same package, they can directly access each other's package-access methods and data variables. They only need to use the reference.variable to do so.

```
int minute; // minute declared without public
// or private.
Time2.minute // Other members of its class can
// directly access minute.
```

4.3 Creating a package

A package is a way to organise classes. Normally, you create a public class. If you don't define your class as public, then it's only accessible to other classes in the same package. Use the keyword package followed by the location.

```
package com.sun.java;
```

The package statement must be the first statement in your class file. Compile it in DOS using the –d option

```
javac –d Time2.java
```

4.4 Final instance variables

The principle of encapsulation is built around the idea of limiting access to variables. This "least privilege" concept can be expanded to include variables that should never be changed, or "Constants". Since this value is a constant, the compiler could optimise by replacing references to that variable with

its constant value. Then there is no need to look up its value when it is referenced.

If a variable is defined as being "final" then it must be initialized in the same statement. It can never again be changed. By custom, it should be declared as all upper case. Any internal words in the identifier should be separated by underscores. If you try to change a variable that you earlier declared as being "final", then the compiler will complain. Obviously, it is better to have the compiler complain, than to have your program crash in production. Whether or not a variable is final is independent of its access. I.e., it can be declared either public or private.

```
private final int NUMBER_OF_MONTHS = 12;
final String FIRST_MONTH = "January";
```

4.5 Final methods and classes

When a method is declared to be final, Java knows that the method can never be overridden. A final method must be fully defined when it is declared. You cannot, for example, have an abstract final method. Since it is final, and can never be overridden by subclasses, the Java compiler can replace the call to the method with inline code if it wants. As for security, if you declare a method to be final, then you can be sure it isn't overridden. For example, class Object has a method called getClass() that is declared final. No subclass can override this method and thereby return some other class type to hide its identity.

A final class makes all of its methods final as well, since a final class cannot be extended. Examples of final classes are: Integer, Long, Float and Double. None of these "wrapper" classes can be subclassed. String is another class that's declared final. So, if you want to stop programmers from every making a subclass of a particular class, you can declare that class to be final.

4.6 Objects passed by reference

As we know, the name or reference for an object represents a memory location where the object is stored. When an object is passed, only the reference is passed. That means, only the address of the object is passed. A copy is *not* made of the object. This will have interesting implications.

Up until now, our classes have either been Applets or Applications. Now we create a class that is neither. This class cannot execute unless it is instantiated by either an Application or an Applet. First we create the class. Then we create another Applet/Application class to test it.

4.7 Types of methods

The different kinds of methods are:

`Constructor` — instantiates the class.

`Accessor` — accesses the member variables in the class.

`Mutator` — (also called Manipulator)—to change or write to the member variables of the class.

`Utility` — private methods that do work for the other methods of the class.

4.8 Our new class

Now we have a new class `Time1`. Class `Time1` keeps time variables. The class also validates any time we wish to create. It prevents us from creating an impossible time. As of yet, there is no actual object. Right now, it is only a recipe for an object of our type. In other words, we haven't instantiated it yet. We cannot instantiate it in this class. We need to create another class to do actually make an example of this class. We need to create a driver program `TimeTest.java`. The only purpose of this new class is to instantiate and test our new class `Time1`.

```
import javax.swing.JOptionPane;
public class TimeTest
{
public static void main( String args[] )
{
Time1 t = new Time1(); // calls Time1 constructor
}
}
```

Now we are using the class `Time1` that we just created. This line creates a reference "t" to an object of type Time1. Then, the "new Time1() calls the Constructor method to instantiate our new Time1 object "t".

```
import javax.swing.JOptionPane;
public class TimeTest
{
public static void main( String args[] )
{
Time1 t = new Time1(); // calls Time1 constructor
String output;
output = "The initial universal time is: " +
t.toUniversalString() + "\nThe initial time is: " +
```

```
t.toString() + "\nImplicit toString() call: " + t;
System.exit( 0 );
}
}
```

In this next line, you see how we are calling two different methods of our new class. We're calling toUniversalString() and toString(). One curious thing is this naked reference " t " sitting out here by itself. What does that do? Well, anytime you concatenate an object to a String, you automatically call its toString() method.

4.9 Varieties of Methods: details

In applications we have created so far, there has only been the method main. In the applets we have created, there have been several standard methods: `init()`, `start()` and `paint()`. Applets have other standard methods, as we know. The class we created `Time1` is neither an application nor an applet. Time1 cannot execute unless another program first instantiates it. Encapsulating data and the methods used to access that data is the central role of the class in object oriented programming. With this class, we make the member variables private. We create methods used to access the member variables. There are several different varieties of methods, and each kind has a different sort of job to do.

Constructors—public methods used to initialize a class.

Accessors—public methods (gets) used to read data.

Mutators—public methods (sets) used to change data.

Utility—private methods used to serve the needs of other public methods.

Finalizers—protected methods used to do termination housekeeping.

The Constructor is named exactly the same as the class. It is called when the new keyword is used. It cannot have any return type, not even void. It instantiates the object. It initializes instance variables to acceptable values. The "default" Constructor accepts no arguments. The Constructor method is usually overloaded. The overloaded Constructor usually takes arguments. That allows the class to be instantiated in a variety of ways. If the designer neglects to include a constructor, then the compiler creates a default constructor that takes no arguments. A default constructor will call the Constructor for the class this one extends.

```
public Time2()
{
setTime( 0, 0, 0 );
```

```
}
```

This is the first Constructor. Notice that it takes no arguments, but it still sets the instance variables for hour, minute and second to consistent initial values of zero.

```
public Time2()
{
setTime( 0, 0, 0 );
}
public Time2( int h, int m, int s )
{
setTime( h, m, s );
}
```

These are the first two Constructors. The second one overrides the first. The second Constructor takes arguments. It still calls the setTime() method so it can validate the data.

Accessors are a public method to display private variables. The access to private member variables is provided through the Accessor, or "gets" methods. This allows the designer of the class to control how and anyone can access the private data. They are also called Query methods. Public methods used to change private variables. Mutators "set" private data. The designer can filter he incoming data and ensure it is correct before it is used to change the private data. This permits the data to be correctly validated before it is used to update the member variables.

• You were sitting in your Ferrarri in your driveway.

• Next door, your plumber neighbor was sitting in her Ferrarri.

• If you wanted to refer to your neighbor's Ferrarri, you would naturally say "Jane's Ferrarri...."

• Likewise, it would be perfectly natural for you to refer to the car you were sitting in as "this Ferrarri...."

4.10 Finalizer methods

We know that the Constructor method is used to instantiate and initialise an object in memory. To avoid the problem called a "memory leak" one which plagues the C/C++ environment it is beneficial to have an anti-Constructor method. Such methods exist. They are called Finalizers, and they control the orderly removal of objects from memory. When an object has gone out of scope, the finalizer executes automatically to clean up (release) used system resources. When there are no longer any references to an object, it is eligible

for garbage collection. Garbage Collection is done automatically to return RAM memory back to the system—so called "termination housekeeping."

We say that an unused object is marked for garbage collection. A finalizer method must always be called finalize() It always takes no arguments and returns void. The finalize() method is one of the 11 methods inherited from method Object.

4.11 Static class methods

When we instantiate a class, each instantiation of that class gets its own private copies of the instance variables for that class. However, in certain cases, we would like to have all the instances of the class share one copy of a variable, instead of each having their own copy.

```
interestRate
```

Say we had 30,000 instances of a class called SavingsAccount. If we changed the interestRate variable they all contained, we would have to make 30,000 method calls to make the change in all of them. However, if we just had defined the interest variable as static, we would have only a single copy that all of them shared.

```
static interestRate
```

Only the members of the class could access the static variable. Instance variables that are defined as being static have class scope. If you define your static instance variables as public:

```
public int static interestRate
```

then this variable can be reached by a reference to any object of that class, or through the class name using the dot operator:

```
savacct.interestRate
```

or

```
SavingsAccount.interestRate
```

Or

```
ClassName.staticFieldName;
```

If you define your static instance variables as private:

```
private int static interestRate
```

Then the private static instance variables can only be accessed through methods of the class, like any other private variable. Static Class members can be accessed even when no instances of that class exist, To access a public static class member when the class is not instantiated, you tack the name of the class to the name of the variable:

```
SavingsAccount.interestRate
```

Math.PI is a static member variable

To access a private static class member when the class is not instantiated, you still prefix with the class name, but then you also use a public static method:

```
SavingsAccount.getInterestRate()
```

When even a single instance of the class SavingsAccount exists (is instantiated), then any of those existing classes can access our static variable (interestRate) simply by using its name: interestRate. When no objects of class SavingsAccount exist, our static variable can still be referenced, but only by going through the Class name and a public static method:

```
SavingsAccount.getInterestRate()
```

Just like static class members (data variables), belong to the class not any one instantiation of the class. Static class methods do not operate on any instance of a class. That means you can use them without creating any instance of a class. For example, all the methods built into the Math class are static methods. You use this general syntax when using :

```
ClassName.staticMethod( parameters );
```

Because static methods do not work with an instance of a class, they can only access static fields. Let's think about this: No instance of the class is instantiated. In this situation, the only thing that could possibly be present is a static member. Thus, the only thing that would be around for a static method to see is another static member. Finally, consider the most famous of all static methods:

```
public static void main( String args[] )
```

Since main is static, you don't need to create an instance of the class in order to call it—and the Java interpreter doesn't either. Once again, "static" means: variables and methods that belong to a class but not to any particular object of the class.

5 Object Oriented Programing (oop)

Although we write the programs in a general way, the objects themselves cause the behavior to be specific. This lets us write programs that will work correctly even for objects that haven't been invented yet.

5.1 Inheritance

We have already been using the terms Superclass and Subclass. A Superclass is the parent class while a Subclass is the child class. A Subclass inherits the behaviour and variables of the Superclass. A Subclass inherits all the methods and all the encapsulated data variables of its Direct Superclass.

Therefore in the following code JOptionPane inherits all the methods and all the encapsulated data variables that lie within JComponent.

```
+--java.awt.Component
      |
      +--java.awt.Container
            |
            +--javax.swing.JComponent
                  |
                  +--javax.swing.JOptionPane
```

Class JOptionPane inherits from class JComponent.

But wait a second. Isn't JComponent the Subclass of Container? Correct. So, that tells us JComponent inherits all the methods and all the encapsulated data variables of Container. Container is the Superclass. So, when JOptionPane inherits everything from JComponent (which itself inherited everything from Container), that must mean JOptionPane inherits everything from both JComponent and Container. So, let's put this together: the JOptionPane inherits all the methods and all the encapsulated data variables in all the Superclasses above it.

javax.swing
Class JOptionPane

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JOptionPane
```

61.png - JComponent is the Superclass, JOptionPane is the Subclass

To understand how this happens consider this question - When we want to create a copy of some object, what do we do? We use the new keyword to fire off the Subclass object's Constructor and we instantiate the object. And so, if we

want to create a copy of the Superclass, we need to somehow, fire off the Superclass object's Constructor.

Whenever you extended another class, (a Superclass) you have created an instance of every class above your class in the class hierarchy. When the Constructor for Sales fires, it silently also fires the Constructors for JApplet and every other class above JApplet in the hierarchy.

```
public class Sales extends JApplet
{
...
}
```

When the Constructor for Sales fires, it silently also fires the Constructors for JApplet and every other class above JApplet in the hierarchy.

The Subclass adds to what it inherits from the Superclass, because we say it adds, that implies it already has everything else. By the time your Subclass is created, it already has an instance of every Superclass above it and every single thing the Superclasses contain. Every method, every data variable present in the Superclass is also present in the Subclass. The point to note here is that if your class has everything above it then you can say it is an example of the classes above it. Finally, since the Subclass has everything the Superclass has, we can say that every Subclass is also an object of the Superclass.

So, let's ponder this: JOptionPane inherits from JComponent. The JOptionPane inherits everything JComponent possesses. So, at the same time we can say JOptionPane is a JComponent, JOptionPane is a Container, JOptionPane is a Component and JOptionPane is an Object.

Classes - Object, Component and Container are Indirect Superclasses to the JOptionPane. JComponent is the Direct Superclass to the JOptionPane. We say JOptionPane is a JComponent, a Container, Component, and an Object.

5.2 Impact of access modifiers

Access Modifiers are among the thorniest and most confusing aspects of OOP. But, remember, encapsulation is one of the priMaya benefits of object orientation, so access is important. The familiar private access modifier lets us shield our instance variables from the prying eyes of the outside

world. Relying on Access Modifiers, you can shield both your class's private instance variables and its private methods. With those ends in mind, Java lets review the four different levels of access: private, protected, public and the default if you don't specify - package.

Specifier	Class	Subclass	Package	World
Private	X			
Package	X		X	
Protected	X	X	X	
Public	X	X	X	X

As you can see, a class always has access to its own instance variables, with any access modifier. The second column shows that Subclasses of this class (no matter what package they are in) have access to public (obviously) and protected variables. Therefore, the important point is that Subclasses can reach protected-access variables, but can't reach package-access variables unless the Subclasses happen to be saved in the same package. The third column "package" shows that classes in the same package as the class (regardless of their parentage) have access to data variables. In the fourth column, we see that anything and anyone has access to a public data variable or method—defeating the purpose of encapsulation.

5.2.1 Private

When a class inherits from a Superclass, you cannot access the Superclass's private data variables. The private data is Secret, In other words, the Subclass cannot automatically reach the private data variables of the Superclass. A private data variable is accessible only to the class in which it is defined.

Objects of type Alpha can inspect or modify the iamprivate variable and can call privateMethod, but objects of any other type cannot.

```
class Alpha
{
private int iamprivate;
public Alpha( int iam )
{
iamprivate = iam;
}
```

```
private void privateMethod()
{
iamprivate = 2;
System.out.println(“” + iamprivate);
}
}
```

The Beta class, for example, cannot access the iamprivate variable or invoke privateMethod on an object of type Alpha because Beta is not of type Alpha .

```
class Beta
{
public void accessMethod()
{
Alpha a = new Alpha();
a.iamprivate = 10; // Invalid
a.privateMethod(); // Invalid
}
}
```

The compiler would complain if you tried this. So can one instance of an Alpha object access the private data variables of another instance of an Alpha object? Well yes objects of the same type have access to one another's private members.

5.2.2 Package

If you do not specify the access for either a method or an encapsulated data variable, then it is given the default access: package. This access allows other classes in the same package as your class to access its data variables as if they were their own. This level of access assumes that classes in the same package as your class are friends who won't harm your class' data.

Notice that no access modifier is declared. So, iamprivate and method privateMethod both default to package access. All classes declared in the package Greek, along with class Delta, have access to iamprivate and privateMethod.

```
package Greek;
class Delta
{
```

```
int iamprivate;
void privateMethod()
{
System.out.println("privateMethod");
}
}
```

If you use multiple objects from the same package, they can access each other's package-access methods and data variables directly, merely by referencing an object that has been instantiated. With package access, other objects in the package don't have to bother going through the methods. They can get right to the variables. So, when you don't include any access modifier, you are in fact giving your variable package access.

```
int x; // package access instance variable
Notice, it's declared neither public nor private.
```

5.2.3 Protected

Protected allows the class itself, Subclasses and all classes in the same package to access the members. Generally speaking, protected offers greater access than package access. Use the protected access level when it's appropriate for a class's Subclasses to have access to the member, but not for unrelated classes to have access. Protected members are like family secrets—you don't mind if someone in the family knows—but you don't want outsiders to know.

The Access Modifier protected can be used on either a method or an encapsulated data variable. Protected serves as a middle ground between the private and public access modifier. A Superclass's protected data variables may be accessed only by: methods of the Superclass, methods of the Subclass or methods of other classes in the same package.

To summarise protected members have package access. Protected access means the variable is public to the class, private outside the class, and public to the subclasses. It's like the Subclasses can reach up and get access to the data variables of the Superclass.

5.3 Super reference

Already, we know that the Superclass-Subclass interaction has a big effect on the Constructor. The Superclass Constructors are either implicitly or explicitly called first in the Constructor of a Subclass.

```
super( whatever the Superclass needs)
```

Using this super reference, you can call the Constructor of your Subclass' Direct Superclass. If you use this super reference, then it must be the very first statement in the Subclass' Constructor.

```
public class Employee
{
String name;
String SSN;
public Employee( String nm, String soc )
{
name = nm;
SSN = soc;
}
public double calc_pay()
{
return 0.0;
}
}

public class HourlyEmployee
extends Employee
{
double hourly_rate;
int hours;
public HourlyEmployee()
{
// implicit (hidden) call to Superclass Constructor.
hourly_rate = 0.0;
hours = 0;
}
public HourlyEmployee( String n, String s, double r,
int h )
{
// implicit (hidden) call to Superclass Constructor.
hourly_rate = r;
hours = h;
}
public double calc_pay()
{
```

```
  return hours * hourly_rate;
  }
  }

   public class HourlyEmployee
  extends Employee
  {
  double hourly_rate;
  int hours;
  public HourlyEmployee()
  {
  super( n, s ); // Explicit call to Superclass Constructor
  hourly_rate = 0.0;
  hours = 0;
  }
  public HourlyEmployee( String n, String s, double r,
int h )
  {
  super( n, s );n // Explicit call to Superclass Constructor
  hourly_rate = r;
  hours = h;
  }
  public double calc_pay()
  {
  return hours * hourly_rate;
  }
  }
```

There is a similar statement that is used to call your Subclass' constructor within your Subclass. Don't get the two confused.

5.4 Subclass as an object of the Superclass

A Subclass contains more than its Superclass contains. Because of that, we say inside of a Subclass is a complete copy of its Superclass. Previously, we said a Subclass is an object of the Superclass type. Now, we take that one step further. A Subclass contains everything (and more) required to make a complete example of the Superclass. Thus, we say an object of a Subclass can be treated as an object of its Superclass. In other words, in certain situations,

we can ignore the difference between the Superclass and Subclass objects.

An object of a Subclass can be treated as an object of its Superclass. What would stop this from being true? Is there any part of the Superclass that is missing from the Subclass? Well no it's all there. Certainly if we are asking a Subclass to fill in for its parent, we throw away the extra stuff that the Subclass added, but—because a Subclass "is an" example of its Superclass—we can "treat the Subclass as an example of the Superclass." And, if we only want to call the methods from the Superclass and access the data variables that come from the Superclass—what's the difference?

Doing this might be interesting. As long as we treat the reference like it only refers to the Superclass, we can handle a Subclass object from a Superclass "reference." As long as we only call the methods that exist in the Superclass, this will work fine.

```
public class Employee
{
String name;
String SSN;
public double calc_pay()
{
return 0.0;
}
}

public class HourlyEmployee
extends Employee
{
double hourly_rate;
int hours;
public double calc_pay()
{
return hours * hourly_rate;
}
}

public class SalariedEmployee
extends Employee
{
```

```
double monthlySalary;
public double calc_pay()
{
return monthlySalary;
}
}

 public class TestEmp
{
public static void main( String args[] )
{
double money = 0.0;
HourlyEmployee hour;
hour = new HourlyEmployee();
money = hour.calc_pay();
System.out.println( "Money=" + money );
}
}

public class TestEmp
{
public static void main( String args[] )
{
double money = 0.0;
SalariedEmployee salr;
salr = new SalariedEmployee();
money = salr.calc_pay();
System.out.println( "Money=" + money );
}
}


public class TestEmp
{
public static void main( String args[] )
{
double money = 0.0;
Employee empl;
```

```
empl = new SalariedEmployee();
money = empl.calc_pay();
System.out.println( "Money=" + money );
}
}
```

As long as we only call the methods that exist in the Superclass, this will work fine.

```
public class TestEmp
{
public static void main( String args[] )
{
double money = 0.0;
Employee empl;
empl = new SalariedEmployee();
money = empl.calc_pay();
System.out.println( "Money=" + money );
}
}
```

So, why in the world might we want to do this?

```
public class RetiredEmployee
extends Employee
{
double monthlyPension;
public double calc_pay()
{
return monthlyPension;
}
}
```

Imagine if after we built our system we decided to add a class, for Retired Employees. If we had built our program on a Superclass reference, then we wouldn't have to rebuild anything. The runtime environment sees what kind of object we have instantiated, and calls the right method override.

```
public class TestEmp
{
```

```
public static void main( String args[] )
{
double money = 0.0;
Employee empl;
empl = new RetiredEmployee();
money = empl.calc_pay();
System.out.println( "Money=" + money );
}
}
```

Caution, this works only when you're calling a method that exists in your Superclass, not one that exists only in the Subclass. And so now you see the advantage of identifying a bunch of empty methods in your Superclass. And the whole process is called, Polymorphism.

5.5 Polymorphism

We can create a Superclass reference that is an array. Then, when we instantiate the array, we can attach all different kinds of Subclass objects to the Superclass array reference. As long as we're only calling methods that exist in the Superclass, we can work our way through the array and it will call all the correct overridden versions of each individual Subclass object. The Superclass reference only knows about the methods that exist in the Superclass. The Superclass only tries to call the methods it knows about. That's perfectly fine, because all the methods in the Superclass are available in its Subclass.

Therefore the Superclass isn't aware that the object it references can do a whole lot more than the Superclass thinks it can. So, "We can create a Superclass reference array that actually points to a Subclass object." As long as we treat those Subclass objects as if they were Superclass objects, (meaning we only call the methods in the Superclass) we have no problems. The Subclass knows how to do everything its parent Superclass can do. The kids can do everything the parent can.

However, if we try to do it the other way around, treating the Superclass as if it were one of its children then we can have problems. The Subclass can do many things the Superclass cannot. The kids can do many things the parent cannot. If we called the Superclass with the Subclass's reference, then you might expect the Subclass reference can do all the things the kids can and you'd be wrong. If you want to go the other way, attach a Superclass object to a Subclass reference, you have to do an explicit cast, as a way of informing the

compiler that you really want to do this dumb thing.

A Superclass object is not a Subclass object. An HourlyEmployee is an Employee (Subclass is a Superclass object) but an Employee is *not* an HourlyEmployee (Superclass is *not* a Subclass object). If you want to actually call some of the methods that don't exist in the Superclass, then you have to first cast the object back to a reference of its own kind.

5.5.1 Constructors and finalisers in subclasses

Whenever you instantiate a Subclass object, the Constructors for the Superclass should be called to initialise the instance variables of the Superclass. This instantiation can be done either implicitly or explicitly (via the super reference). These calls to instantiate the Superclass object must the first thing in your class's Constructor method. The opposite of the Constructor method is the finaliser method. If your class bothers to define a finalizer method, then the implicit or explicit call to the Superclass finaliser must be the last thing in the finalizer method. You will always want to define your finaliser method as protected, so that Subclasses that inherit from your class can access its finalizer. Classes that only use Superclass objects (not inherit them) cannot use the Superclass's finalizer methods.

You must always use this "super" reference when you intend to call the finaliser of the Superclass. If you omit the super class, you're actually calling the finaliser of the method you're already inside of – which is called infinite recursion. You cannot do a cascading call to a Superclass. Instead, you have to rely on the Superclass itself to call its own Superclass.

5.5.2 Implicit Subclass-object to Superclass-object conversion

In order to understand the concept of subclass and superclass object, let's consider the following illustration. Ravi was feeling entrepreneurial, and so he decided to buy a piece of land in Bihar. Ravi paid ₹50 for what the deed described as a wooden shack in the swamp. But, to his great surprise, when he got there, Ravi discovered he hadn't bought a shack in a swamp, but rather, the Taj Mahal on a beautiful lake. Ravi was ecstatic. Ravi got more than he was promised. Maya was feeling entrepreneurial too, and so she decided to buy a huge mansion on an estate in Bihar. She paid ₹5 Million for what the deed described as a palatial estate on 40 acres of land. But, to her great surprise, found she hadn't bought a magnificent mansion and estate, but rather, a Doublewide trailer next to a dump. Maya was horrified. Maya got less than was promised.

Ravi's situation is akin to referring to a Subclass object with a Superclass reference. You get more than you expected. It's a pleasant surprise.

```
SuperclassRef = new SubclassObject
```

Maya's situation is akin to a Superclass object with a Subclass reference. You get less than you expected. It's a tragic mistake.

```
SubclassRef = new SuperclassObject
```

There are four possible conversions that can occur:

1. Refer to a Superclass object with a Superclass reference. This is routine
2. Refer to a Subclass object with a Subclass reference. This is also routine
3. Refer to a Subclass object with a Superclass reference.

This is safe because the Subclass object is an object of its Superclass. Such code can only refer to Superclass methods. You can make an array of Superclass references. Then, you can treat them the same, as if they really were all Superclass objects. As long as you call methods that exist in the Superclass (but were overridden by each of the various Subclasses), then the runtime system calls the correct overridden method for each on. This process is known as Dynamic Binding.

4. Refer to a Superclass object with a Subclass reference.

This is a syntax error! It doesn't make sense. It is only remotely possible if the Subclass is explicitly cast into a Superclass reference, which is a way of telling the compiler you are doing this damn-fool thing with your eyes open. Here's the problem: A Subclass contains more and does more than its Superclass parent. If you use a Subclass reference to point to a Superclass object, you are implying that the object has more data variables available than it really does—and if someone made that assumption—they would be sorely mistaken.

5.5.3 Use Polymorphism

```
class Employee
{
private String ssn;
public Employee()
{
ssn = "";
}
public Employee( String soc_num )
{
ssn = soc_num;
```

```
}
public String getSSN()
{
return ssn;
}
public double calcPay()
{
return 0;
}
}
```

Employee is our Superclass. As we see, it contains the instance variable String ssn, a Constructor, the Accessor method getSSN() and the empty method calcPay(), which we will need to override in all the Subclasses because they will all need to calculate pay differently.

```
class HourlyEmployee extends Employee
{
double hourlyRate;
int hoursWorked;
public HourlyEmployee()
{
hourlyRate = 0.0;
hoursWorked = 0;
}
public void setHourlyRate( double rate )
{
hourlyRate = rate;
}
public void setHoursWorked( int hours )
{
hoursWorked = hours;
}
public double calcPay()
{
return hourlyRate * hoursWorked;
}
}
```

```
public class DemoPolymorphism
{
double pay = 0;
public DemoPolymorphism()
{
Employee supEmp = new HourlyEmployee();
HourlyEmployee subHrEmp = new HourlyEmployee();
pay = supEmp.calcPay();
System.out.println( "Superclass pay = " + pay );
subHrEmp.setHourlyRate( 5.65 );
subHrEmp.setHoursWorked( 20 );
pay = subHrEmp.calc_Pay();
System.out.println( "Subclass pay = " + pay );
supEmp = subHrEmp;
pay = supEmp.calcPay();
System.out.println( "Superclass pay = " + pay );
}
public static void main( String[] args )
{
DemoPolymorphism demo = new DemoPolymorphism();
System.exit( 0 );
}
}
```

We declare a reference to an object of type Employee called `supEmp`. However, we instantiate an HourlyEmployee object and assign the reference to the Superclass Employee. This works because—after all—HourlyEmployee is an Employee.

```
public class DemoPolymorphism
{
double pay = 0;
public DemoPolymorphism()
{
Employee supEmp = new HourlyEmployee();
HourlyEmployee subHrEmp = new HourlyEmployee();
pay = supEmp.calcPay();
```

```
System.out.println( "Superclass pay = " + pay );
subHrEmp.setHourlyRate( 5.65 );
subHrEmp.setHoursWorked( 20 );
pay = subHrEmp.calc_Pay();
System.out.println( "Subclass pay = " + pay );
supEmp = subHrEmp;
pay = supEmp.calcPay();
System.out.println( "Superclass pay = " + pay );
}
public static void main( String[] args )
{
DemoPolymorphism demo = new DemoPolymorphism();
System.exit( 0 );
}
}
```

```
Superclass pay = 0.0
Subclass pay = 113.0
Superclass pay = 113.0
Press any key to continue . . .
```

The Wrong Way to Use Polymorphism—a Syntax Error—(Without An Explicit Cast)

```
public class CastSubTypeReftoSuperTypeObject
{
double pay = 0;
public CastSubTypeReftoSuperTypeObject()
{ // Syntax Error ! ! !
HourlyEmployee subHrEmp = new Employee(); // Error!
// The error is not yet revealed.
pay = subHrEmp.calcPay();
System.out.println( "Subclass pay = " + pay );
// Error because the object doesn't have these
// methods available.
subHrEmp.setHourlyRate( 5.67 );
subHrEmp.setHoursWorked( 20 );
pay = subHrEmp.calcPay();
```

```
System.out.println( "Subclass pay = " + pay );
}
public static void main( String[] args )
{
CastSubTypeReftoSuperTypeObject demo;
demo = new CastSubTypeReftoSuperTypeObject();
System.exit( 0 );
}
}
```

In the program, we know that subHrEmp was declared as a type HourlyEmployee. But instead of getting a Mansion (HourlyEmployee), the reference was assigned to a Doublewide Trailer (Employee). The object doesn't have these methods and these message calls won't work!

```
C:\CastSubTypeReftoSuperTypeObject.java:59:
Incompatible type for declaration.
Explicit cast needed to convert Employee to HourlyEmployee.
HourlyEmployee subHrEmp = new Employee(); // Error!
^
1 error
Tool completed with exit code 1
```

At this point, the compiler is extremely unhappy. So, let's give it what it is specifically asking for an explicit cast and see what happens then.

```
public class CastSubTypeReftoSuperTypeObject
{
double pay = 0;
public CastSubTypeReftoSuperTypeObject()
{ // Now we've Band-Aided the Error with An
// Explicit cast
HourlyEmployee subHrEmp;
subHrEmp = (HourlyEmployee) new Employee();
// The error is not yet revealed.
pay = subHrEmp.calcPay();
System.out.println( "Subclass pay = " + pay );
// Error because the object doesn't have these
// methods available.
```

```
subHrEmp.setHourlyRate( 5.67 );
subHrEmp.setHoursWorked( 20 );
pay = subHrEmp. calcPay();
System.out.println( "Subclass pay = " + pay );
}
public static void main( String[] args )
{
CastSubTypeReftoSuperTypeObject demo;
demo = new CastSubTypeReftoSuperTypeObject();
System.exit( 0 );
```

Exception in thread "main" java.lang.ClassCastException: Employee at astSubTypeReftoSuperTypeObject.<init>(CastSubTypeReftoSuperTypeObject.java:60)atastSubTypeReftoSuperTypeObject.main(CastSubTypeReftoSuperTypeObject.java:75)

Press any key to continue . . .

At this stage, the compiler is incensed, it trusted us when we asked for that Explicit cast, and look what happened.

5.6 Polymorphism and dynamic binding

Say that we have a bunch of objects that are all Shapes: Circles, Squares, Triangles, Trapezoids, etc. If we want to draw them all, we could use a large switch statement that asks if the object is a circle, do this. If a square, do this. If a triangle, do that. The big switch could work okay, but if we decided to add a parallelogram to our shapes, then we would have to remember to add another branch to our switch statement. Every place we wanted to do some behavior based on the shape, we would have to change our logic.

Polymorphism can save us that trouble and make programs that will work with Shapes that hadn't even been thought of when the class was written. With Polymorphism, the objects themselves know what kind of object they are, and they do the right behavior based on their internal awareness of their own type. In our Shapes example, we could assume that each shape Subclass that inherits off of the master Superclass—Shape—would have a draw() method. That draw() method would detail exactly how to generate the Square or the Circle or the Triangle. Each shape subclass knows how to draw itself. Certainly, we could go through the trouble of drawing each Shape one by one. But wouldn't it be nice if we could just treat all those various Shape Subclasses as copies of the Superclass? Then, we could create an array of Superclass references that

actually are initialized with Subclass objects. We could move through the array calling the draw() method of the Superclass for each element in the array. The overridden draw() methods in each Subclass shape could take care of drawing the shape the right way.

Therefore, to make this work, the only thing we have to remember is, Declare draw() in the Superclass, then, override draw() in each of the Subclasses to draw the correct Shape. The Run-Time Type Identification system will choose the correct subclass's draw() method dynamically i.e., on-the-fly at execution time. This is called dynamic method binding, and it makes Polymorphism work. Polymorphism relies on Dynamic Binding. Polymorphism, allows you to process Superclass objects in a Generic way. Only the parts of a program that need specific knowledge of the new class must be tailored specifically to the new class.

6 Arrays and strings

An Array consists of data items of the same type, all with the same name. The size of an Array is "static". Once you have created an "array", you cannot change the number of elements it holds. If you create an array with 42 occurrences, then it will always have 42 elements. An array is a Data Structure. If you wish to create an array where the number of elements can be changed, you use class Vector. In Java, an array is a group of contiguous memory locations that all have the same name and same type.

This declares that x is an array of char variables.

```
char[] x;
```

Allocating the array decides the number of elements.

```
x = new char[5];
```

If you remember "x" is a reference, the syntax above is more understandable. When you allocate an array, the elements are automatically initialized.

Primitives:

numeric primitives are zeroed,

char primitives are made spaces,

boolean primitives are made false

References:

For an array of any other type the "references" are made null.

```
x[0]= 'Z';
```

This assigns the char Z' to the first array position.

```
x[1]= 'q';
```

This assigns the char 'q' to the second array position.

An array is a full-fledge object. You can declare an array either of two ways:

```
char[] x;  or char [];
```

If you put the brackets on the data type, you can declare multiple references as all being the same type of array,

```
char[] x, y, z;
```

In this case, x, y and z are all declared as char arrays.

To learn the number of elements in the array, you call a property called "length." For example:

```
int y = 0;
int[] x;
x = new int[5];
y = x.length;
```

Notice, "length" is *not* a method, it is a property. You can use an "Initializer List" enclosed in braces to quickly insert values in your array:

```
int n[] = { 12, 44, 98, 1, 28 };
```

Because this list has five values in it, the resulting array will have five elements. If we had a char array, it would be like this:

```
char m[] = { 't', 'I', 'M', 'Y' };
```

6.1 Constant variables and the final qualifier

When used in a variable's declaration, the keyword "final" indicates that the variable can never be changed.

```
final int ARRAY_SIZE = 10;
```

If you have chosen to make your variable "final" then you must initialize it in the same statement that declares it--you have no other alternative.

When we call a method and pass the method a primitive-data-type variable, we are always passing a copy of the original, not the original. Thus, if the method changes the variable it receives, only the copy is changednot the original.

```
int x = 3;
```

x is a primitive variable. If we call a method with x as an argument, we make a copy of x and the original copy is not affected by any changes we make in the copy.

```
int x = 3;
yMethd( x )
public int yMethd( int y )
{
return y++;
}
```

If we pass x to a method, a copy y is made and any changes happen to the copy. Now, we have `char[] out = {'H','e','l','p'};`

If we pass the reference out to a method, we again pass a copy, but the copy points back to the original, and so our method can change the original.

6.2 Passing an array to a Methasfod

As you saw in the previous example, when we wish to pass an array to a method, the array's signature must be expecting to receive an array. Thus, the method was declared:

```
public void strMethd( char[] data )
{
}
```

The method was clearly designed to receive a char array, which it names data. But when we called the method, we only used the bare name of the array we were sending. We didn't use the square brackets.

```
strMethd( out );
```

If we pass an entire array – meaning just the array name without any square brackets – then we are passing the reference and the original array can be changed in the method. But, if we pass just one array value (and include the square array brackets) then we are just passing a copy and the original cannot be changed. If we pass just the naked array name, the original array can be accessed and changed.

```
char[] out;
...
wholeArray( out )
public void wholeArray( char[] data )
{
data[2] = 'L';
// will change original array
}
```

If we pass just a single array value, we can't change the original table.'

```
char[] out;
...
pieceOfArray( out[2] )
public void pieceOfArray( char d )
{
d = 'x';
// won't change original array
}
```

```
public void start()
{
char[] out = { 'H', 'e', 'l', 'p' };
display.append( "Before out=" + out[0] + out[1] );
wholeArray( out );
```

```
pieceOfArray( out[1] );
display.append( "\nAfter out=" + out[0] + out[1]
+ out[2] + out[3] );
}
public void wholeArray( char[] data )
{
char change = 'l';
data[3] = change;
display.append( "\nIn wholeArray: " + data[0] + data[1]
+ data[2] + data[3] );
}
public void pieceOfArray( char datum )
{
datum = 'W';
}
```

The call using the entire array was able to change the original. The call using just a single element was not able to change the original.

6.3 Double-Subscripted arrays

This is the syntax for a double-subscripted array

```
char demo[][]
```

Java does not directly support two-dimensional arrays--however, it lets you define an array where each element is an array--thereby achieving the same effect.

6.4 Strings and characters

Strings do a lot of things we have not been aware of.

```
public class Test
{
public static void main( String[] args )
{
String w = "Welcome to ";
String j = "Java ";
float v = 1.2f;
String txt;
txt = w + j + v + ".";
System.out.println( txt );
```

```
}
}
```

Note – There are 3 String variables referenced here and 5 blocks of character storage are allocated?

```
public class Test
{
public static void main( String[] args )
{
String w = "Welcome to ";
String j = "Java ";
float v = 1.2f;
String txt;
txt = w + j + v + ".";
System.out.println( txt );
}
}
```

Why does such a tiny program need so many areas of memory? Let's explore the answer to that question in detail. All Java characters are stored in 16-bit Unicode, so they are able to store international character sets. In Unicode, every character requires 2 bytes. Under this system, "Hello, world." would require 26 bytes to store in memory.

To create Strings, we do something like,

```
String myname = "Joe";
//initialized to Joe
```

or

```
String myname = new String("Joe" );
// initialized to Joe
```

The bottom method is less familiar, but more correct under the Object-Oriented method. The top method is a convenience we use in order to make String manipulation easier. Another convenience is in String concatenation:

```
String a = "hello ";
String b = "world";
String c;
```

```
c = a + b;
```

We accept that we're not arithmetically adding these two Strings. Rather, the plus [+] operator has been overloaded. "Overloading" is when something has a different meaning depending on its context. When we do this: c = a + b; it seems like we are really doing this: c = a.concat(b); In fact, secretly, we're using another class called a StringBuffer, which we'll get to a little later.

Take the following example, where we are concatenating a String a with an int i:

```
String a = "Ten ";
int i = 4;
String c ;
c = a + i;
```

Since i is being "added" to a String, the compiler knows it needs to convert i to a String also. The compiler creates a new String block just to hold the converted integer. Anytime you concatenate a String with another type, the other type is first converted into a String and then the two are concatenated. Really, this is done using method toString()which every object inherits from Object

```
c = a.concat( (String) b );
```

If you concatenate other primitive data types with a String, it has a similar effect:

```
String a = "";
boolean b = false;
String c ="Microsoft is dishonest=";
a = c + b;
Microsoft is dishonest=false
The += operator is also overloaded:
String c = "Microsoft ";
String b = "rules";
c += b;
{ c == "Microsoft rules" }
```

The following example is a crackerjack Certification question:

```
String a = "";
int b = 2;
int c = 3;
a = b + c;
```

This is a syntax error. To force the conversion to String, at least one of the

operands on the + sign must be a String. What about this example, where we're concatenating a String object to a regular object. What happens here?

```
String a = "";
String b = "Test";
Employee e = new Employee("GW","Bush");
a = b + e;
```

This still produces a String, and is equivalent to:

```
a = b + e.toString();
```

Class String is located at the top of the class hierarchy. The class is declared as final, which means it cannot be Subclassed, cannot be a Superclass to any Subclass. Class String contains 48 methods. These methods allow you to do many things with a String object except change the String. You can do anything with a String except change that String. There are many methods that take a String argument and return a String. None of these allow you to change the original String. These methods may appear to change the String, but they don't. They merely return a different String. Once a String object is instantiated, it can never be changed. Strings are immutable. That means, once they are created, they can't be changed. Because String objects are immutable they can be shared.

Here we have three String objects, a, b and c, which happen to contain identical Strings. Because they are identical, the Java compiler only stores one copy of the String "hello".

```
String a = "hello";
String b = "hello";
String c = "hello";
```

a, b and c are merely pointers to the same space.

Once the compiler discovers that the String object is the same for all three, it uses the same String for all three. Remember, a String can't be changed. In the following example, we are reassigning a reference, or making it point to a different place. We're not changing the String:

```
String a = "hello";
String b = "hi";
String c;
c = a;
c = b;
```

This helps explain what is happening when we compare two Strings in this manner: if(a == b); We are comparing the references, not the objects.

6.4.1 Length of a string

To find out the length of a String or the number of characters a String has you use the String's method length():

```
String a = "Testing";
int x = 0;
x = a.length(); {x == 7}
```

To find the length of an array object, you use the property length. Notice, it's not a method. To find the object , you use the method length().

6.4.2 Sub-strings

You can extract a substring from a larger String object with the substring() method of class String.

```
String greet = "Howdy";
String s = greet.substring( 0, 4 );
```

The first argument 0 is the 1st character of the substring that you do want to copy. The 4 is the 1st character that you don't want to copy.

```
So...{s is equal to "Howd"}
```

6.4.3 Finding individual characters

The String function charAt() allows you to discover and return the character at a certain point in a String.

```
String a = "Testing";
char b = ' ';
b = a.charAt( 3 );
{ b == „t }
```

6.4.4 Comparing strings

You now know you cannot compare the references of two String objects in order to determine if the referenced objects are equal. You accomplish that goal using the equals() method of class String.

```
String a = "hello";
String b = "hello";
a.equals( b );
```

In fact, you can even take a shortcut to the same compare:

```
String b = "hello";
"hello".equals( b );
```

In fact, you can even take a shortcut to the same compare:

```
String b = "HELLO";
"hello".equalsIgnoreCase( b );
```

```
String b = "hacker heaven";
String n = " ";
n = b.replace( 'h', 'H' );
```

Remember, this does not change the original, it return a new String object in which the changes have been made.

```
String b = " hacker heaven ";
String n = "";
n = b.trim();
{ n == "hacker heaven" }
```

This method merely removes any extra spaces from both the front and back of a String. Note that this does not alter the original String. Rather, it returns a new String with the leading and trailing spaces omitted.

6.4.5 StringBuffer Class

An object of class String cannot be altered after it has been created. An object of class StringBuffer can be altered. We describe the class StringBuffer as a mutable class, meaning it can be changed. StringBuffer are used internally to implement many of the methods in the String class. Every StringBuffer object can hold only a certain amount of characters. A StringBuffer has a capacity that describes the number of characters it can hold. This number is determined at the time of creation. If not specified when it is instantiated, a StringBuffer has a default capacity of 16 characters. Therefore, you generally specify the capacity when you instantiate your StringBuffer. We say a StringBuffer is Dynamically Resizable. A StringBuffer is instantiated as follows:

```
StringBuffer d = new StringBuffer();
```

This one will begin with a capacity of 16 characters.

```
StringBuffer s = new StringBuffer( 100 );
```

This one will begin with a capacity of 100 characters.

```
StringBuffer e;
e = new StringBuffer( "hello" );
```

This one has a capacity of 21 characters, because the initializing String contains 5 characters.

Every StringBuffer starts off with 16 characters of capacity. When you initialize it with a String, you still get the original 16, and your initializing String is just added to that 16. If you add characters to your StringBuffer, it grows in size, it's Dynamically Resizable. You don't worry about its size, it's a question of efficiency.

For example, this StringBuffer would be inefficient:

```
StringBuffer e;
e = new StringBuffer( 99999999 );
```

This one will hold 99999999 characters. If you do wish to add to your StringBuffer , you use its append() method

```
StringBuffer e;
e = new StringBuffer( 100 );
e.append( "Key Largo" );
```

A StringBuffer makes your code run faster because you create fewer new objects such as temporary Strings. The method append() always adds the characters to then end of the StringBuffer. The method append() is overloaded 10 times. append() takes arguments of every primitive data type and will convert each to a String automatically. You cannot intermingle String methods with StringBuffer methods.

When the overloaded plus sign [+] concatenates a String to something else, it uses a StringBuffer object:

```
d = "Ten" + 4 + "ya";
```

The above is actually implemented as follows:

```
d=new StringBuffer().append("Ten").append(4).append("ya");
```

This is initially created with the default capacity.

Method append() is the principle way to add characters to the end of a StringBuffer. StringBuffer method insert() is used to place characters into an existing StringBuffer object at a specified location. Method insert() is overloaded 9 times.

Whereas append() adds text to the end of the StringBuffer object, method insert() takes two arguments. The first argument is an integer "offset" that tells the first character position to begin inserting the second argument. The second argument can be any data type.

Classes String and StringBuffer do share a few methods. You can learn the length of a StringBuffer by using the method: length(). Method length() tells you the amount of memory a StringBuffer currently is using. Method

capacity(), on the other hand, tells you the total amount of memory allocated to a StringBuffer.

Method reverse() simply reverses the order of the characters in the StringBuffer.

```
StringBuffer king;
king = new StringBuffer( "murder" );
king.reverse();
{ king.toString() == "redrum" };
```

Method ensureCapacity() simply makes sure that your StringBuffer has at least the capacity you specify. This is primarily an issue of efficiency. One could say the argument of this method is something called "Minimum capacity."

```
ensureCapacity( int minimumCapacity );
```

There is a twist on how you apply ensureCapacity()

```
StringBuffer sb;
sb = new StringBuffer("Test");
```

First (Certification-type question) What is the current capacity of StringBuffer object sb?

20 = 16 + 4

Since we know sb has a current capacity of 20, let's try a few experiments:

```
StringBuffer sb;
sb = new StringBuffer("Test");
sb.ensureCapacity( 19 );
```

What would be the capacity of sb after this statement?

20 = 16 + 4

Wat would be the capacity of sb after this statement?

```
StringBuffer sb;
sb = new StringBuffer("Test");
sb.ensureCapacity( 21 );
```

If you asked for less that was already allocated, nothing changes. However, if you asked to increase the allocation, it gives you the larger of two-times the original plus 2 OR your request.

42

Finally, let's test that:

```
StringBuffer sb;
sb = new StringBuffer("Test");
sb.ensureCapacity( 43 );
```

Since this request is larger than two-times the original allocation, plus

two, our request prevails.

43

6.5 Character class

Most methods in the Class Character are Static. Most of these methods take a character argument. They perform either a test or a manipulation on that argument.

method isDefined() seeks to know if the character that is provided as an argument is defined in the Unicode character set.

```
int x = "}";
Character.isDefined( x );
```

true means the character is defined.

This method tries to determine if a character is an alphabetic letter of any language in the Unicode character set or if it is not.

```
int x = "t";
Character.isLetter( x );
```

true means the character is a letter.

This method tries to determine if a character is a digit.

```
int x = "3";
Character.isLetter( x );
```

true means the character is a digit.

6.6 StringTokenizer class

The StringTokenizer class is a tremendously useful and tricky class that can be used to do things like break a String up into words. What we call a "word", the computer knows as a "Token"or a unit of a String. You will find this term is used commonly in other languages also.

The item used to decide where one token ends and another begins is called the delimiter. Commonly, you choose the delimiter to be a space, tab or newline. Although these are the common delimiters, you get to choose what you want to use for a delimiter. In fact, a token can be many characters, not just one.

In fact, you can change the delimiter every time you call a method of the StringTokenizer class object. When you use an object of class StringTokenizer, there is one primary decision that has to be made. If the returnTokens flag is false, then the delimiter characters only separate other tokens. If the returnTokens flag is true, then the delimiter characters are themselves tokens. Remember, a token doesn't have to be just one character,

it can be many characters that together comprise the token.

As the StringTokenizer class object moves its way through a String, it automatically knows where it is in the String, you do not have to keep track of that. When you instantiate an object of type StringTokenizer, you have three alternatives arguments for the Constructor

```
1.) String stringtobetokenized
2.) String stringtobetokenized,
String delimiter
3.) String stringtobetokenized,
String delimiter,
boolean returnTokens
Default delimiters: " \n, \t, \r "
Default returnTokens = false.
```

6.6.1 StringTokenizer Class: Methods

```
int         countTokens()
boolean     hasMoreTokens()
boolean     hasMoreElements()
Object            nextElement()
String            nextToken()
String             nextToken( String delimiter )
```

countTokens() method counts how many more times this tokenizer object's nextToken can be counted.

```
int countTokens()
```

It returns an integer with that number.

hasMoreTokens() simply tests if there are more tokens available from this tokenizer's String.

boolean hasMoreTokens() method if returns true, it anticipates that the a call to method nextToken() right after this would succeed in returning a token.

hasMoreElements() is used to override a method inherited from the Enumeration interface, which this class implements.

```
boolean hasMoreElements()
```

In effect, this does the same thing as the hasMoreTokens() method. true if there are more tokens.

nextToken() method is the workhorse. It gets the next token waiting in the String that was used to instantiate this object.

`String nextToken()`

If you neglected to precede this call with a successful hasMoreTokens() method call, this method can throw a NoSuchElementException. This method overloads nextToken(). It still returns the next token in the String but

`String nextToken( String delimiter )`

If you neglected to precede this call with a successful hasMoreTokens() method call, this method can throw a NoSuchElementException.

7 Files and streams

Before you can read from a file, you must open it. After you are done reading from a file, you must close it. There are two common varieties of reading:

reading characters (a character is 16 bits long)

reading bytes (a byte is 8 bits long)

Inevitably, when you sit down to read from a file, you have to sort through the choices on those lists. The best approach is to pick one from either list—character and byte—and learn to use it.

7.1 Reading Characters from a File

Say you want to read from a file:

- You will need to open the file.
- You will need to read from the file to the end.
- You will need to close the file.

First of all, what is a file? A file is an instance of the class File

```
import java.io;
public class FileRead
{
public FileRead()
{
File inFile = new File( "C:/orig/aFile.txt" );
}
public static void main( String[] args )
{
FileRead fr = new FileRead();
}
}
```

All the classes used in I/O come from this package. Because the constructor on File throws an IOException, we are forced to place it in a try-catch block:

```
import java.io;
public class FileRead
{
public FileRead()
{
try
```

```
{
File inFile = new File( "C:/orig/aFile.txt" );
}
catch( IOException io )
{
System.out.println( "IOException, io=" + io );
}
}
public static void main( String[] args )
{
FileRead fr = new FileRead();
}
```

Now an example to read characters from a file, and store it as integers.

```
public class FileRead
{
public FileRead()
{
try
{
File inFile = new File( "C:/orig/aFile.txt" );
File outFile = new File( "C:/final/outFile.txt" );
FileReader fr = new FileReader( inFile );
FileWriter fw = new FileWriter( outFile );
int c = 0;
boolean keepReading = true;
while( keepReading )
{
c = fr.read();
if( c == -1 )'
{
keepReading = false;
}
else
{
fw.write( c );
}
```

```
}
fr.close();
fw.close();
}
}
}
```

7.2 Reading bytes from a file

The approach for reading bytes is nearly the same. The difference comes in the classes we choose to do the reading and writing.

```
public class FileRead
{
public FileRead()
{
try
{
File inFile = new File( "C:/orig/aFile.txt" );
File outFile = new File( "C:/final/outFile.txt" );
FileInputStream fis = new FileInputStream( inFile );
FileOutputStream fos = new FileOutputStream( outFile );
int c = 0;
boolean keepReading = true;
while( keepReading )
{
c = fis.read();
if( c == -1 )'
{
keepReading = false;
}
else
{
fos.write( c );
}
}
fr.close();
fw.close();
}
}
```

```
}
```

7.3 Alternatives for efficiency

As you can imagine, reading a byte or a character at a time is pretty inefficient. For that reason, there are alternatives. The best one is the BufferedReader. This class gathers a chunk of data at a read. The BufferedWriter also allows us to write an entire String.

```
public class FileRead
{
public FileRead()
{
try
{
File inFile = new File( "C:/orig/aFile.txt" );
File outFile = new File( "C:/final/outFile.txt" );
FileReader fr = new FileReader( inFile );
BufferedReader br = new BufferedReader( fr );
FileWriter fw = new FileWriter( outFile );
BufferedWriter bw = new BufferedWriter( fw );
String temp = null;
boolean keepReading = true;
while( keepReading )
{
temp = br.readLine();
if( temp == null)
{
keepReading = false;
}
else
{
bw.write( temp );
}
}
br.close();
fr.close();
bw.close();
fw.close();
}
```

```
}
}
```

7.4 File serialization

Another variant of the file I/O world is something called serialization. To serialize an object means to take an object in memory and write that object to a file. Then, at a later time, the object can be de-serialized and then we have the object—with its state intact—back in memory.

```
String myString = new String( "Some important text" );
File myFile = new File( "C:/myFile.ser" );
FileOutputStream out = new FileOutputStream( myFile );
ObjectOutputStream s = new ObjectOutputStream( out );
s.writeObject( myString );
s.flush();
```

To start, we create an object of type String "myString". Next, notice we are using a special class called an ObjectOutputStream. This class is designed to serialize objects. By custom, files of serialized objects end in ".ser". Finally, we see that we are writing an object.

The process to read from an existing Serialized file is very similar.

```
File myFile = new File( "C:/myFile.ser" );
FileInputStream in = new FileInputStream( myFile);
ObjectInputStream s = new ObjectInputStream(in);
String myString = (String)s.readObject();
```

Notice, when you read out the object from serialized file, you need to cast the object back into the type you know it is.

7.5 Reading user input from the console

Although it should be easy, you have to consider the user inputting values from the console as reading from a stream. Before we look at the program, let's understand the issues involved:

- Now do we tell it to read?
- How do we tell it to stop reading?

You tell it to read a line by hitting the "return" key on your keyboard. To tell it when to stop reading, we need to send in a "sentinel" value. That means, no matter what, stop when you read this "sentinel" value.

```
String temp = null;
boolean keepReading = true;
InputStream is = System.in;
```

```
  InputStreamReader isr = new InputStreamReader( is );
  BufferedReader br = new BufferedReader( isr );

  StringBuffer stuffRead = new StringBuffer();
  try
  {
  while( keepReading )
  {
  temp = br.readLine();
  if( temp == null || temp.length() == 0 )
  {
  keepReading = false;
  }
  else
  {
  stuffRead.append( temp);
  }
  }
  System.out.println( "stuffRead=" + stuffRead.toString()
);
  }
  catch( IOException io )
  {
  System.out.println( "ConsoleReader Constructor threw an
IOException, io=" + io );
  }
```

Here, we see that an entry of spaces is the sentinel value.

8 Multithreading

8.1 Thread

It is the basic unit of program execution. A process can have several threads running concurrently, each performing a different job. When a thread has finished its job, it is suspended or destroyed.

Threading distinguishes Java from C++, and is the ability to have multiple "simultaneous" lines of execution running in a single program. If you are still stuck with a computer that has only one CPU, it's a myth that separate threads run simultaneously. Because they are sharing the cycles of the one CPU, each thread is given a priority – which is a number from 1-10. (5 is the default priority.) Threads with a higher priority number are given more turns at the CPU. In fact, depending on the operating system, a thread's priority can have a more drastic impact.

Depending on the OS, one of either two strategies is used:

- preemption or
- timeslicing.

Under Sun's Solaris operating system, a thread of a certain priority is allowed to run until it is finished, or until a thread of higher priority becomes ready to run.

When a thread of higher priority arrives and steals the processor from a lower-priority thread, this is known as preemption. One weakness of the preemption system is this: if another thread of the same priority is waiting to run, it is shut out of the processor. This is called thread starvation.

Another system is used in the Windows world. In a process known as timeslicing, each thread is given a period of time to run, and then all the other threads of equal priority get their turn at the CPU. As with the preemption system, under timeslicing, a higher-priority thread still commandeers the CPU immediately.

8.2 Class Thread: an Overview

`java.lang.Thread`

When we wish to make a Java program use threading, the easiest way is to say our class extends the class Thread. The Java Class Thread has 7 different constructors.

1.Thread()—this constructor takes no arguments.

Every instantiated thread object has a name. When we use the no-argument

Constructor, the names are automatically generated. A thread created using the Constructor above would be named:

Thread-0, Thread-1, Thread-2, etc.

To instantiate a Thread using this default Constructor, we would do like this:

Thread t1 = new Thread();

t1 would have the thread name of Thread-0.

2.Thread(String name)

This Constructor takes a String argument, which becomes the thread's name.

```
String n = "cece";
Thread t2 = new Thread( n );
```

t2 would have the thread name of cece.

Similarly,

```
String n = "cece";
Thread t3 = new Thread( n );
Thread t4 = new Thread();
```

t3 would have the thread name of cece.

t4 would have the thread name of Thread-0.

(There are 5 more Constructors with more ambitious intentions. For now, these two are sufficient.)

Once we have instantiated a thread object, we'd like to put it to work. All of the work done by a thread happens in the method:

run()

This method takes no arguments and returns void.

Normally, you will never call the run() method yourself. You will call another method— start() —and the JVM will call run() when it is good and ready.

As you would expect, we begin by extending the class Thread.

```
public class MyThread extends Thread
{
}
```

First, add the constructor method—which names the thread "Thread-x". Now, because it sub-classes Thread, class MyThread is an instance of the thread class. Class Thread contains a method run() that is empty—it does nothing. So, if you want anything to happen in your thread, you need to

override the method run().

```
public class MyThread extends Thread
{
public MyThread( String n )
{
super( n );
System.out.println( “MyThread Constructor” );
}
}
public void run()
{
System.out.println( “In run!\n” );
}
```

Next we build a driver program and put in a method main. When main executes, the JVM places it in a single thread context.

```
public class MyThread extends Thread
{
public MyThread( String n )
{
super( n );
System.out.println( “MyThread Constructor” );
}
public void run()
{
System.out.println( “In run!\n” );
}
}
public class TestMyThread
{
public static void main( String[] args )
{
MyThread chuck = new MyThread( "Charlie" );
chuck.start();
MyThread dede = new MyThread( ”Deirdra" );
dede.start();
MyThread pat = new MyThread( “Patrick" );
pat.start();
```

```
}
}
```

main() will instantiate and start() all these threads. After they have started, the methods immediately return and the thread main() dies. However the entire application doesn't die until the last thread does. By default, a new thread has the same priority as the thread that created it.

Notice what happened here—in our main method, we instantiated our MyThread class. Then, we executed the start() method of our object. We never called the run() method but, lo and behold, it got called.

So, how many threads did we give life to in that example?

4

main

chuck,

dede,

pat

So, using threads is much like running a road race:

Before running, you must start

1.Understanding the sequence:

My program is running along happily. Without me doing anything, my program is already running in a thread—main. (Therefore, your programs have always been running in a single thread.)

2.In this context—where a single thread (main) is already running—you're going to execute the following code:

```
public static void main( String[] args )
{
MyThread mt = new MyThread();
mt.start();
}
```

Firstly, understand that your goal is to start up a new thread. When you instantiate a new thread object, and then execute its start() method, you are asking the JVM to create a new, additional thread to run alongside the thread you created when your original main() method started up.

We say that start() launches a new thread, and then returns to the calling program, thereby permitting the calling program to continue working independently of the just-launched thread.

Although you have a method run() in your MyThread class, don't ever execute the run() method yourself. Let the JVM do it for you after you execute

the start() method.

```
public static void main( String[] args )
{
MyThread mt = new MyThread();
mt.start();
}
mt.run();
```

Question: what would happen if you yourself executed the method run() in your MyThread class?

Answer: you would execute the run() method in the same thread—thread main—that you were already in, not in a new thread.

What will be the output when we execute main() of the following code?

```
public class MyThread extends Thread
{
public MyThread( String name )
{
super( name );
}
public run()
{
System.out.println( "In run()-thread=" + Thread.
currentThread() );
}
}
public static void main( String[] args )
{
MyThread chuck = new MyThread( "Charlie" );
chuck.start();
MyThread dede = new MyThread( "Deirdra" );
dede.start();
MyThread pat = new MyThread( "Patrick" );
pat.start();
}
```

Answer:

In run()-thread=Thread[Charlie,5,main]

In run()-thread=Thread[Deirdra,5,main]

In run()-thread=Thread[Patrick,5,main]

So, pretend you are holding the starter pistol for a road race. You

start the threads, but let them do the running. If you try to execute a start() method on a thread that has already been started, the start() method throws an

IllegalThreadStateException

8.3 static Thread Methods

Many thread methods are static : A static method belongs to the class—not to any one instance of the class. Therefore, you can execute a static method even if you haven't instantiated the class—such as the case of main.

8.3.1 static void sleep(long millisecs)

Causes the currently executing thread to sleep (meaning temporarily cease execution) for the specified number of milliseconds.

During this sleep time the thread does not use any resources of the processor.

8.3.2 static Thread currentThread()

Returns a reference to the currently executing thread object.

In other words, if you want to know which thread is currently executing, you execute this method.... and what would be the syntax to do that?

```
Thread x;
x = Thread.currentThread();
```

8.3.3 static void yield()

Causes the currently executing thread object to temporarily pause and allow other threads of equal priority to execute.

You would use the method yield() only if your operating system did not support preemptive multithreading.

8.3.4 static boolean interrupted()

True or False. Tests whether or not the current thread has been interrupted recently.

This static method discovers whether or not the current thread has been interrupted. Calling this method also resets the "interrupted" status of its argument. This particular method can be easily confused with another method similarly named. The emphasis here is on CURRENT THREAD. For this method, we don't get to choose the thread we wish to test. By default, it tests whether or not the current thread is interrupted.

8.3.5 boolean isInterrupted()

Tests whether or not *this* thread has been interrupted. This method can be used to discover whether any thread has been interrupted.

Also, calling this method does *not* change the interrupted status of its argument.

8.3.6 void interrupt()

Interrupts this thread. If called on a thread object that is already currently blocked, the blocking call (such as sleep() or wait() complains about the interrupt() call and terminates with an InterruptedException.

Sends an interrupt message to a thread. The "interrupted" status of the thread is set to true. This doesn't ask a question, it performs an action

8.4 void wait()

The thread itself calls the wait() method to wait for a specific condition to be satisfied.

8.4.1 boolean isAlive()

returns true if:

start() has been called for the thread, and
the thread is not yet dead, meaning
the run() method has not yet finished and died.

8.4.2 String getName()

It returns the thread name for the thread.

8.4.3 void setName(String n)

It sets the thread name for the thread.

8.4.4 static void dumpStack()

It causes a dump of the execution stack at this instant.

```
public class MyThread extends Thread
{
public MyThread( String n )
{
super( n );
System.out.println( "MyThread Constructor" );
```

```
}
}
public class TestMyThread
{
public static void main( String[] args )
{
MyThread pat = new MyThread();
pat.start();
Thread.dumpStack();
}
}
```

We start off with our typical class MyThread that extends Thread. We then instantiate a thread object, and execute the start() method. Now, because the thread method dumpStack() is static, we call it on the class object.

```
MyThread Constructor
java.lang.Exception: Stack trace
java.lang.Throwable(java.lang.String)
java.lang.Exception(java.lang.String)
void java.lang.Thread.dumpStack()
void TestMyThread.main(java.lang.String [])
```

After dumpStack()

You see main() was the first thing executed. You read a stack dump from the bottom up, the way it was stacked. Next, Thread's static method dumpStack() was called. Finally Exception was called, followed by Throwable, the Superclass.

8.4.5 final void setPriority(int p)

This method allows you the change the priority to an integer between 1 and 10.

8.4.6 final int getPriority()

This method lets you return the priority for a thread. Notice, this method is final to prevent you from causing any mischief.

8.5 Thread states

Life Cycle of a Thread:

• born - Thread is Instantiated. When start() method is called, thread enters ready state.

- ready - Highest priority thread that is ready enters running state
- running - System assigns a processor to the thread and the thread begins to execute its run() method. When run() method completes or terminates, thread enters dead state.
- dead - Thread marked to be removed from the system. Entered when run() terminates or throws uncaught exception.
- blocked - To enter blocked state, thread must already have been in running state. Even if the CPU is available, a blocked thread cannot use the processor. Common reason for a thread being in a blocked state—waiting on a request for file I/O .
- sleeping - Entered when sleep() method has been called for a thread. While in the sleep state, a thread cannot use the processor. After its sleep time expires, the thread re-enters the ready state. It doesn't starting running again, it becomes eligible to run—much in the same way it did when you called the start() method.
- waiting - Entered when wait() method has been called in an object thread is accessing. One waiting thread becomes ready when object calls the method notify(). When the method notifyAll() has been called—all waiting threads become ready.

A thread executes until:

- it dies,
- it calls sleep(),
- it calls wait(),
- it calls yield(),
- its quantum of time is used up, or
- it is preempted by a higher priority thread.

When a thread has just been instantiated, your only option is to start() it. If you try to execute any method besides start(), you will cause an

IllegalThreadStateException

When the start() method is returns the thread is ready to be run when the scheduler decides it's the right time. Notice on the right there is the state called "not runnable." There are several reasons why a thread that has not finished its run() method could find itself "not runnable." :

- If a thread has been put to sleep, then the sleep milliseconds must elapse.
- If a thread is waiting for something, then another object must notify the waiting thread of a change in condition. The other object would handle the notification by calling the methods notify() or notifyAll().
- If a thread is blocked on I/O, then the I/O must complete.

Let's make sure we have that sequence straight:

- Thread zelda is waiting for something else.
- Because it cannot go on without the something it's waiting for, zelda calls the method wait() on itself: zelda.wait();
- Thread zelda remains in the waiting state until the object it is waiting, say scott, announces it is ready by issuing the notify() or notifyAll()method.

8.6 Thread synchronization

Until this point, all of our threads have lived alone. However, it's possible to have threads share information. In a class, it is common for many threads to all have access to instance variables that have class scope. However, this usually raises issues that could never arise when threads were not involved. For example, what happens when two threads try to write to or read from an instance variable at the same instant?

Coping with this problem raises the thorny subject of thread synchronization.

```
public synchronized int getBalance()
{
...
}
```

The first piece of the puzzle is a thing called a monitor. When we say an object is a monitor, we are merely saying that object has the ability to pay attention to the behavior of threads. We accomplish thread synchronization by making methods synchronized. Further, we say every object that contains synchronized threads is in effect a monitor. So in sumMaya, if an object is a monitor, then that object permits only one thread at a time to execute a synchronized method on that object.

This is implemented as follows:

- whenever a synchronized method is invoked, the object is locked.
- If my object has a synchronized method, and something calls that synchronized method, then my object is locked up tight.

In fact, for every Java object that contains a synchronized method, Java creates a lock associated with that object.

```
public class LockBox
{
private int contents;
private boolean available = false;
public synchronized int get()
```

```
{
return contents;
}
public synchronized void put( int value )
{
contents = value;
}
}
```

Note that methods get() and put() are each synchronized. If my class were calling the put() method, then nobody could call either the put() or the get() method in that instance. I have locked up that instance and it's hands off for everybody until I'm done. Because the class LockBox contains a synchronized method, every instance of class LockBox would have a special lock associated with it.

So, you see: when somebody is putting a value to the contents variable, nobody else can get it until the put is done. Likewise, when somebody is getting the value currently in the contents variable, nobody else can change that value in the middle of the read.

In other words, if an object has more than one synchronized method, then only one of those synchronized methods at a time can be executing. Synchronized methods are like having a single bathroom—no matter how many need it, it can only be used by one person—boy or girl—at a time. When the synchronized method is done executing, it releases the lock. The monitor gives the go ahead to the highest-priority thread that is ready to execute the synchronized method. If a synchronized method gets its turn to execute but then—during that execution—it finds it cannot proceed (maybe I/O is blocked or there's some other hold up) —then the thread may voluntarily call its method wait(). In the wait() state, the thread doesn't take up processor resources and doesn't hold a lock on the monitor.

When a thread executing a synchronized method completes, the thread can execute the notify() method. Executing the notify() method tells any waiting threads to become ready so the scheduler can get them back running again. The `notify()` method is like calling:

"Head up, you're on deck."

After a thread has been notified, it can attempt to get a lock on the object. Similarly, if a synchronized method no longer needs its lock for some reason, it can call notifyAll(), which alerts all waiting threads that they are free to enter the ready mode and again attempt to get a lock on the object.

Warning: if a thread is waiting, it may wait forever—in a state called deadlock—unless some other thread notifies it that it can enter the ready state. So, if you ever make something wait() be sure to wake it up!

When an object is a monitor, that means it keeps a list of all threads that wish to execute synchronized methods. Just as a method alone can be synchronized, it is possible to just place curly brackets around a chunk of code with the keyword this as an argument. That means, while the tiny block of code so isolated is executing, the object referred to by the this is locked.

Be Careful: synchronization should be used like a spice, not a gravy. It's overuse can spoil your application and slow it way down.

9 Graphics and 2D in Java

9.1 GUI in Java

When Java 1.0 was first rolled out, it only contained the abstract window toolkit (AWT), for basic GUI programming. The original AWT lacked many features that one would expect in an object-oriented toolkit:

- clipboards,
- support for printing and
- keyboard navigation.

The AWT even lacked popup menus and scroll panes – something modern GUI developer could live without. In a word, the AWT was practically useless. And you would ask yourself why? The original AWT deals with GUI elements by not dealing with them. It delegated their creation and behaviour to the native GUI toolkit on each target platform. We can say the underlying "Peer" text box actually handled the GUI. Peers are native GUI components that are manipulated by the AWT classes.

You want to create an instance of the Menu class, the Java runtime system creates an instance of the menu peer. The menu peer does the real work of of displaying and managing the menu behaviour.

Under "peer" architecture, the peer – the underlying operating system takes care of painting windows and reacting to events. The AWT is like a lazy construction supervisor who attempts to look busy while he delegates all the real work to the harried construction worker bees (peers). The ease of peer architecture allowed the early developers to crank out AWT "components" that really didn't do much. But this style of architecture means that each platform's native window actually renders the window, and from the diagram, you can see why we call them "heavyweight" a lot was happening. And this also means slow performance. Every peer on a different operating system produced a different result. Therefore, if you used AWT on a Windows machine, your GUI looked like Windows. If you used AWT on a Macintosh, your GUI looked like Macintosh. And if you used AWT on an X11/Motif machine, your GUI crashed and burned. (X11/Motif machines could not support complex graphical elements such as menus and scrollbars.) So, using the original AWT GUI, you found that your application looked completely different on the various platforms and, what was worse, it encountered different bugs on different platforms, such as the ones on Motif. These bugs came from mismatches between the peers

and the AWT components. Developers were forced to test their applications on each platform, a practice that became infamously known as write once, debug everywhere.

A year after Java 1.0 came out, Netscape created a Java GUI library called the IFC (Internet Foundation Classes) that used a fresh approach. The IFC was a set of lightweight classes based on ideas from NEXTSTEP's User Interface Toolkits. These did not rely on peers. Javasoft, fearing that the Java community would fracture because of the new IFC, made a deal with Netscape to adapt the IFC. Using the IFC approach, User Interface Components (buttons, menus, textboxes—called Widgets) were painted onto blank windows. The only peer functionality needed was a way to put up windows and paint on the window. Netscape's IFC widgets looked and behaved the same no matter which platform they ran on.

Sun accepted Netscape's lead on this process and created a new user interface library named after a Duke Ellington song:

"It Don't Mean A Thing If It Ain't Got That Swing..."

Swing is now the official name for the non-peer-based GUI toolkit that is part of the Java Foundation Classes (JFC). Because Swing was not part of the original core Classes, but rather, was an extension, it was named javax.swing. Thus, the AWT library is still used, especially the version 1.1 Event Handling library. Swing elements are somewhat slower than their AWT counterparts, so the AWT elements are still valuable. The AWT library of classes is still needed to this day—Swing is built on AWT. But Swing takes advantage of the AWT's infrastructure, including graphics, colors, fonts toolkits and layout managers.

Now you need to remember that Swing does NOT use the AWT's components. Of all the components the AWT has to offer, Swing uses only Frame, Window and Dialog, as the Superclasses to JFrame, JWindow and JDialog. In effect, Swing cherry picked the best of the AWT to build a new set of mostly lightweight components, while discarding the parts of AWT that were the most trouble—the heavyweight components. It was intended to replace the AWT's heavyweight components, but not the AWT itself. So, in order to understand the Swing GUI library (next week), we need to understand the AWT library first.

9.2 Lightweight components

The 1.1 API introduced the notion of lightweight components. Lightweight components directly extend either:

- `java.awt.Component`
- `java.awt.Container`

The Component class and its supporting cast are the foundation upon which the AWT is built. Component is an abstract base class for components such as menus, buttons, labels and lists. And Container is an abstract base class that extends Component. Classes derived from Container are

- Panel,
- Applet,
- Window,
- Dialog and
- Frame.

All can contain multiple components.

9.2.1 Component

It is an abstract class that provides a great deal of functionality for the classes that extend it. For example, a component has the following affiliated with it.

- Graphics object
- Location
- Size
- Native peer
- Parent container
- Fonts and font dimensions (font metrics)
- Foreground and Background colors
- Locale
- Minimum, Maximum and Preferred sizes

A component does not decide for itself how it will be shaped, and so its shape ends up taking a lot of abuse. Now, before you go blaming Container for this abuse, you must know that Container is an innocent bystander. Although a Container contains a Container , it is the LayoutManager who is 100% responsible for the abuse dished out to a Container . The LayoutManager decides the sizing and positioning of components. Using Containers, you can group related components and treat them as a unit. This simplifies an applet's design and is useful for arranging components on the display.

9.2.2 Layout managers

Containers merely keep track of what components they contain. They delegate the positioning and shaping of their components to a layout manager. The LayoutManager interface defines methods for laying out components and calculating the preferred and minimum sizes of their containers. The AWT provide five classes that implement either the LayoutManager or LayoutManager2 interfaces.

- FlowLayout – specifies that components flow left to right, top to bottom.
- BorderLayout – lays out North/South/East/West/Center components.
- CardLayout – displays one panel at a time from a deck of panels.
- GridLayout – lays out components on a simple grid. Components are stretched to fill the grid.
- GridBagLayout – imposes constraints on each component in a grid.

Flow layout

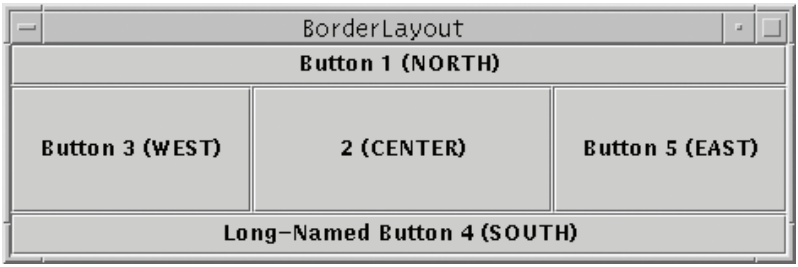
It displays components left to right, top to bottom and is the default layout manager for panels and applets. This just shoves in the components willy-nilly. When an instance of FlowLayout is constructed, the alignment (LEFT, CENTER, RIGHT) can be specified, in addition to horizontal and vertical gaps. The horizontal and vertical gaps specify not only the gaps between components but also between the edges of the components and the sides of the container. (This of course is in direct contrast to the BorderLayout, where the gaps are only between components.)

Border layout

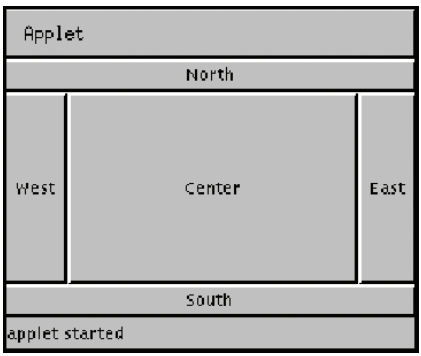
It lays out components around the sides and in the center of the container in North, East, South, West and Center positions. Gaps between components can be specified. It is the default layout manager for Window, Dialog and Frame. BorderLayout is by far the most frequently used layout manager. It is the default manager for most of the AWT containers.

You should use it when a Container divides its components into regions: North and South, East and West. Once you've created a few nested layouts, using panels to join components into sub-panels, you'll appreciate the quick convenience of the BorderLayout, which sets components (or groups of them) into geographical regions. Nearly every nested layout has a BorderLayout lurking somewhere inside.

Here is an example that shows how you can specify the horizontal and vertical gaps between the components:




```
BorderLayout borlay = new BorderLayout( 2, 2 );
```



Card layout

It displays one component at a time from a deck of components. Components can be swapped in and out. You should use it when you want to control the visibility of a set of components under different circumstances or when a set of panels present themselves as a stack of Tabbed Folders.

From this deck of components, you can display or deal any one container at a time. You use this for containers such as a tabbed panel. This layout manager allows clients to control which component is currently displayed. The Card Layout manager is unique in that you can directly control which component is displayed at any one time.

CardLayout Stacking Methods:

- void first(Container) - displays the first component added to the container
- void last(Container) - displays the last component added to the container.
- void next(Container) - displays the component added to the container after the currently displayed component. If the current component is the first, then the last component is displayed.
- void previous(Container) - displays the component added to the container before

the currently displayed component. If the current component is the first, then the last is shown.

• void show(Container, String) - shows the component whose name matches the String passed in. If no components match the name, the method does nothing.

The CardLayout is instantiated with a horizontal gap of 10 pixels and a vertical gap of 5 pixels. Since only one component can be displayed at a time, the gaps specify a margin around the components, rather than between the components, as is the case with the BorderLayout.

Grid Layout

It lays out components in a grid, with each component stretched to fill its grid cell. Horizontal and vertical gaps between components can be specified. You should use it when you need to create something like a calendar or a spreadsheet.

You can set up the number of rows and columns it contains either at the time of instantiation, or anytime thereafter.

```
GridLayout gl = new GridLayout( 3, 0, 10, 10 );
```

GridBag Layout

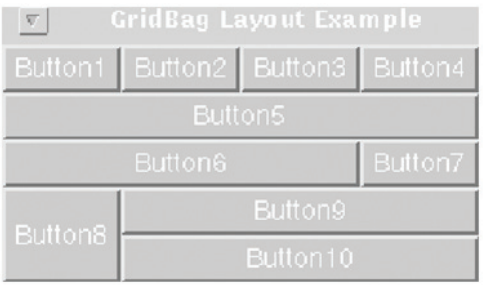
It arranges components in a grid using an elaborate set of grid constraints. You should use it when you're building complicated input forms, such as a container that has components for name, address, city, state and zip.

Unlike GridLayout, where you explicitly specify the number of rows and columns in the grid, the GridBagLayout itself determines the number of rows and columns from constraints placed upon the components it lays out. Also, unlike GridLayout, GridBagLayout allows components to span more than one rigid cell. Components may also overlap.

The GridBagLayout is one of the toughest and most complex in all of AWT. Therefore, we will use divide and conquer to learn it. Lets see an example of this.

```
// Creates a <code>GridBagConstraint</code> object with all of its fields set to their default value
  public GridBagConstraints()
  {
  gridx = RELATIVE;
  gridy = RELATIVE;
  gridwidth = 1;
  gridheight = 1;
```


```
  weightx = 0;
  weighty = 0;
  anchor = CENTER;
  fill = NONE;
  insets = new Insets(0, 0, 0, 0);
  ipadx = 0;
  ipady = 0;
  }
```



To describe the factors that limit the placement of components in a GridBagLayout, we use a special class called the GridBagConstraints. When you are adding components to a container that has been set up with a GridBagLayout, you:

- Set up the constraints in the object of type GridBagConstraints
- Set the constraints for your object using the GridBagConstraints object.
- Add the object whose constraints you just set.

```
JButton butt = new JButton( "Hairy" );
GridBagLayout gbl = new GridBagLayout();
GridBagConstraints gbcon = new GridBagConstraints();
Container c = getContentPane();
c.setLayout( gbl );
gbl.setConstraints( butt, gbcon );
c.add( butt );
```

9.3 The Original Class – Graphics

First we will focus on the original set of Graphical components, and then move on to the modern Graphics2D class that comes with Swing. As we said, the original Java 2D API works on the basis of the AWT, Abstract Window Toolkit, which relies on the underlying hardware.

9.3.1 Graphics basics

The Java coordinate system begins in the upper-left- hand side of the screen. The 0,0 location is actually behind the title bar. You use the method `getInsets()` of class Container to detect where you can start drawing. There are four measurements that tell you how many pixels away from the edge is the available drawing area.

9.3.2 Graphics context

A graphics object remembers a collection of settings for drawing images and text, such as the font you set or the current color. This collection of settings is known as a Graphics Context. All drawing in Java must go through this graphics object, using the established graphics context. Before you can draw on the screen, you have to acquire it. Naturally, a graphics object takes care of acquiring the graphics context, and uses it to control how information is drawn.

A graphics object encapsulates state variables for:

- the component object on which to draw
- information about "clipping" an object, or
- masking pieces of it.
- current color
- current font
- information about the pixels

A Graphics object has methods for:

- font manipulation,
- drawing
- color manipulation.
- Color

This class defines methods and constants used to manipulate colors. Colors are created from:

- red,
- green, and
- blue components.

RGB values: 3 Integers from 0-255 each. [This equates to 16.7 million colours.] Alternately, you can three floating-point values between 0-1.0. The larger the number, the more of that color is added. Alternately, you can use the Color Constants. You pass a Color object to the method setColor()

```
Color col = new Color( 255, 0, 0 );
g.setColor( col );
Or,
g.setColor( new Color( 255, 0, 0 ) );
You get a Color object from the method getColor()
Color x = g.getColor();
Or,
g.setColor( new Color( 255, 0, 0 ) );
```

Swing offers a special class called a JColorChooser—something AWT never offered. The JColorChooser opens a predefined color dialog. This dialog lets you pass arguments that specify which object to change color, a text message and a color object.

9.3.4 Font

The Constructor for class Font takes three arguments:

- font name - The font name is any one currently supported by the system the program is running on.
- font style - The font style is any combination of the well-known alternatives Plain/Italic/Bold. Java supplies Class constants:
- Font.PLAIN
- Font.ITALIC
- Font.BOLD
- font size - The size is measured in points. One point is 1/72nd of an inch.

Example:

```
public class Fonts extends JFrame
{
public Fonts()
{
super( "Using fonts" );
setSize( 420, 125 );
show();
}
```

```
public void paint( Graphics g )
{
g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
}
}
```

This is the usual way. This fires the default constructor with the Font Name, the Font Style and the Font Size. To display text, you must first select a font. To select a font, you must first create an object of the class Font. You specify a font by its font face name (or font name for short), and its point size.

Font cou = new Font("Courier", Font.BOLD, 24);

24 points (A point is 1/72nd of an inch)

9.3.5 Drawing lines and rectangles

Class Graphics has a raft of methods that can be used to draw.

Example - Single line

```
drawLine( int x1, int y1, int x2, int y2 )
```

This will draw a line between the point defined by (x1,y1) and (x2,y2)

Rectangles

```
drawRect( int x1, int y1, int width, int height )
```

This will draw a rectangle with the top-left-hand corner (x1,y1) with the width and height specified.

A rectangle with a filled-in center

```
fillRect( int x1, int y1, int width, int height )
```

This will draw a filled rectangle with the top-left-hand corner (x1,y1) with the width and height specified.

A rectangle with the same color as the background color

```
clearRect( int x1, int y1, int width, int height )
```

This will draw a filled rectangle with the top-left-hand corner (x1,y1) with the width and height specified, with the same color as the background color.

9.3.6 Drawing arcs

An arc is a portion of a circle. It is drawn in an imaginary box, whose size we specify. It has a degree that it starts from, and a number of degrees that it sweeps. Contrary to intuition, a positive number of degrees for the sweep describes an arc that goes counter clockwise. Likewise, a negative number of degrees for the sweep describes an arc that goes clockwise. There are two priMaya methods at work:

```
drawArc( int x, int y, int width, int height, int startAngle, int arcAngle )
```

You must set the color with a different method.

```
fillArc( int x, int y, int width, int height, int startAngle, int arcAngle )
```

9.3.7 Drawing Polygons and Polylines

A polygon is a complex, multisided shape. It is a complex, multi-angled line. This draws a closed shape, even if the locations of the points do not make that occur. So, no matter where you place the points, the shape is closed.

```
drawPolygon( int xPoints[], int yPoints[], int points )
```

drawPolygon also has a constructor that takes another polygon object

```
drawPolygon( Polygon p )
fillPolygon( int xPoints[], int yPoints[], int points )
```

For the fill, you have to issue a separate method call. This draws a series of lines that are connected. In this case, if you do not make sure the last point coincides with the first, then the shape enclosed by the polylines is not closed.

```
drawPolyline( int xPoints[], int yPoints[], int points )
```

9.4 Graphics2D

The awt graphics are used for basic html graphics and are not suitable for complex graphics. For example, Class: Graphics lets you draw only a very few specific shapes:

- rectangles,
- ovals and
- polygons.

Class Graphics2D allows you to draw any shape. Also, Graphics2D , in contrast to the awt package, allows you to decide the style of line you wish to draw, including its width and fill properties—known as a line's texture. To start with, all these Geometric shapes are created starting with interface and Shape. Curves and arcs are created using the same interface. To determine the fill style of an object, you use: interface and Paint. To determine the pen style of an object, you use: interface and Stroke.

First of all, notice that class Graphics is the Superclass to Graphics2D. Every time you have used this method:

```
public void paint( Graphics g )
```

you have in fact been using a Graphics2D object.

Before you can access the Graphics2D capabilities, you must perform a downcast on the Graphics g reference passed to paint. This is safe because—trust me—the object is actually of type Graphics2D:

```
public void paint(Graphics g )
{
Graphics2D g2d = ( Graphics2D ) g;
Class Graphics versus Graphics2D
```

Normally, we would hesitate to cast a Superclass reference to a Subclass type, because we know the Subclass is likely to have many methods not present in the Superclass. However, this is safe merely because we know g is really a Subclass object. Anytime you wish to convert a Graphics reference into a Graphics2D reference, you must use this downcast.

```
public void paint(Graphics g )
{
Graphics2D g2d = ( Graphics2D ) g; d
```

10 Java GUI components

10.1 GUI design

A GUI interface is not designed casually. The designer has to pay close attention to something called "Look and Feel." The guiding principle is making the GUI interface intuitive. To become technically proficient with the important "Swing" API library, you must become comfortable with two base classes and their functions:

- Component - paint() and repaint(), for example, originate in class Component.
- Container - A Container is a collection of related Components.

Swing borrows AWT functionality for:

- graphics,
- fonts and
- layout managers.

Lightweight components are ultimately derived from the AWT's Container class, which extends AWT's Component class. Therefore: Everything in Swing is a Container, is a Component.

10.2 Model-View-Controller architecture

Lightweight Swing components hand off their look and feel to a User Interface [UI] delegate. You could say look equals painting the component. And feel equals handling its events.

UI delegates can be plugged into a component when the component is constructed, or any later time. Thus, the term: "Pluggable look and feel". Swing's pluggable look and feel is based on a component architecture called: Model-View-Controller. The MVC architecture is designed for applications that need to wear many hats.

- Model—used to describe a way to maintain the data and provide access to it via accessor methods.
- View—used to paint a visual representation of some or all of the model's data.
- Controller—used to handle events.

Model stores data, gives a way to access and modify data, fire events to registered Views, so the Views can update themselves based on the change in the underlying data. View takes care of displaying the data. And Controller handle events. ActionListeners and MouseListeners are MVC controllers.

The objects we know as "Listeners" are actually controllers.

MVC is the basis for Swing's Look and Feel. Another advantage of MVC is that any View is automatically updated when anything changes in the Model (data) it is based on. Finally, because the Model is independent of the Views and Controllers, any changes to the Views or Controllers do not affect the underlying Model.

An Example- Button:

The Button is a model—it keeps its name and caption(its Data). The Button doesn't decide how to draw itself—the View decides that using a Layout manager. And neither does The Button to react to being pressed—the Controller decides how to handle that. As you might imagine, the upshot of this design is flexibility. When the data value of a model changes, the Controller hears about it. The Controller [Listener] is notified of the change. An event happens, and the Listener hears about the event.

There are two Kinds of Notification

- Lightweight Notification - Minimal information is passed. This kind of notification is used when a model experiences frequent events. Example: A JSliderBar, passes many change events as the Grip is moved. Because the potential exists for so many events, the information in each is minimized.
- Stateful Notification - The event contains more state information than just the event source. Example: A JButton, passes just one action event as the button is pressed, so more information about the event is passed with each.

```
Model Interface                          Used by
                              Notification
BoundedRange Model                    JProgressBar,JSlider,
JScrollBar                          Lightweight
Button Model                        JButton,    JCheckBox,
JCheckBoxMenuItem,            Stateful
JMenu, JMenuItem, Lightweight
JRadioButton, JRadioButtonMenuItem,
JToggleButton
ComboBox Model                       JComboBox
                    Stateful
```

10.3 JLabel

It used for display only, can display text, an image, or both. It cannot react to events, therefore cannot get the keyboard focus. Basically, it is an area where uneditable text and text with icons are displayed. You can set the

alignment of your JLabel text within the label. By default, text is placed to the right of an image, an image is centered and text alone is left-justified. You can use HTML to specify the text of a label, and thereby get a more specific appearance. You can also set up tool tips, i.e.:

```
myLabel.setToolTipsText( "Display This" );
```

Because this setToolTipsText() method is inherited from class JComponent, then any descendants of JComponent also have this method !

This is how you would place an icon on the JLabel after it was already instantiated:

```
myLabel.setIcon( myIcon );
```

Set alignment:

```
myLabel.setHorizontalAlignment( CENTER );
```

You don't need to repaint the label after you change the alignment, because the change triggers a repaint().

```
myLabel.setIconTextGap( intGap );
```

Integer values allow you set the gap between text and image.

This is an example of a JLabel that display an Icon, and that has had its state enabled and disabled:

Also, you can make the label have a different icon only when it is disabled by:

```
lbl.disabledIcon( Icon icon )
```

10.4 JTextField

Swing provides two distinct types of text components:

Simple text controls that can only display one font, and one color at a time:

JTextField, JTextArea, JPasswordField

Styled text components that can display multiple fonts, and multiple colors:

JEditorPane, JTextPane

JTextField uses the ActionListener interface to respond to events. When a JTextField fires an action event, the event's actionCommand is set to the text field's actionCommand property. Using the JTextField's setActionCommand() method, you can explicitly set this property. Otherwise, it defaults to the contents of the JTextField. By default pressing the ENTER key while a JTextField has focus fires an Action Event. You set the Preferred Width by instantiating it with a number of columns, but the Layout Manager has the final say. The columns you set are assumed to be the letter "m", because it's the widest.

10.5 JPasswordField

A JPasswordField is just another kind of JTextField, except it displays a string of asterisks in place of the text you're typing in. You can make an ActionListener listen to your JPasswordField object, but when it comes time to actually discover what is inside the actionCommand, you must cast your ActionEvent object back into an object of type JPasswordField .

10.6 JButton

Constructors are:

- JButton()
- JButton(Icon)
- JButton(String)
- JButton(String, Icon)

It Is initialized with a String that appears on it and can display an Icon. When the button isn't selected or pressed, you can change the Icon or see how it is currently set.

```
void setIcon(Icon)
Icon getIcon()
```

At run time, you can set or get the text displayed on the button.

```
void setText(String)
String getText()
```

Whenever the button is disabled (grayed out), you can change what icon it displays.

```
void setDisabledIcon(Icon)
Icon getDisabledIcon()
```

You can specify which Icon it will display when it is pressed.

```
void setPressedIcon(Icon)
Icon getPressedIcon()
```

A JButton can display a "mnemonic", or an underline that identifies the shortcut key.

```
void setMnemonic(int)
char getMnemonic()
```

This has two alternative overrides:

1.one accepts a char argument

```
setMnemonic(char mnemonic)]
```

2.The other accepts an int constant of a type KeyEvent.VK_X

```
setMnemonic(int mnemonic)
```

(This one is preferred)

```
void setActionCommand(String)
String getActionCommand(void)
```

setActionCommand() —this lets you define the action command string this button will generate when an event happens to it.

getActionCommand()—this is the command an Event Handler would use to discover the action command string. By default, a button's action command is set to the button text. We have used this method to process events.

When an event occurs, the button passes an ActionEvent object to the listener registered to listen for events on the button. The ActionEvent object has two commonly used methods:

- getSource() which returns the reference name of the object the event happened to, and
- getActionCommand() which "Returns the command string associated with this action."

This usually means, "Return the String displayed on the button," but you can set it to something more specific.

```
b1.setActionCommand("disable");
b3.setActionCommand("enable");
public void actionPerformed(ActionEvent e)
{
if ( e.getActionCommand().equals("disable") )
{
b2.setEnabled(false);
b1.setEnabled(false);
b3.setEnabled(true);
}
else
{
b2.setEnabled(true);
b1.setEnabled(true);
b3.setEnabled(false);
}
}
```

In fact, this is the preferred way to use this setActionCommand() method. Your program sets the Action Command String on an object that has an Action Listener. When the event happens, you can fine tune the result. Remember, when we call the method getActionCommand() of an

ActionEvent object [e.getActionCommand()] it returns a String. If we want to make a comparison between that command String, we can't just compare the two, we have to use the equals() method, which is inherited from Object.

The JButton object lets you specify a different "rollover" icon—when a mouse pointer is on top of the button:

You can set a JButton to be the Default Button, meaning it automatically has the focus. To do this, you have to get an object of the JRootPane, which we have not yet had to do:

```
Container c = getContentPane();
JRootPane root = getRootPane();
JButton def = new JButton( "Default Button" );
root.setDefaultButton( def );
c.add( def );
```

Using a method called doClick(), you can make a button be clicked programmatically—meaning seemingly by magic. The argument is the time in milliseconds that the button will remain pressed.

```
JButton click = new JButton( "Click" );
click.doClick( 2000 );
```

10.7 JCheckBox

These are two-state buttons that reflect their status by having a checkmark in a box (as the name implies). We say they can be Selected or DeSelected.

Constructors are many and varied. It can include a String, an Icon and it can be preset as either selected or deselected. Event Handling for a JCheckBox is different—it uses an ItemListener rather than an ActionListener. Your program must implement the ItemListener interface. This interface has only one method for you to override:

```
public void itemStateChanged( ItemEvent e );
```

An ItemEvent object is passed, and it contains a constant property: ItemEvent.SELECTED

Method getStateChange() tells what happened.

```
JCheckBox ans = new JCheckBox( "Yes" );
ans.addItemListener( this );
…
public void itemStateChanged( ItemEvent e )
{
if( e.getStateChange() == ItemEvent.SELECTED )
}
```

If you instantiate the JCheckBox without an image, it will get the familiar square box. If you instantiate the JCheckBox with an image, then the square box will be omitted and no default square box will be created. When you create a menu, you may also add a menu item that is a check box. For this purpose, you would use a:

```
JCheckBoxMenuItem
```

This control follows the standard L & F as other objects. For example, you can set the Mnemonic as follows:

```
JCheckBox ans = new JCheckBox( "Yes" );
ans.setMnemonic( KeyEvent.VK_Y );
```

This is how to change the status of the control, and of course, this would trigger an itemStateChanged event.

```
ans.setSelected( true );
```

10.8 JRadioButton

These are two-state buttons that reflect their status by having a dot in a circular hole. The interior of the hole is either white or gray, depending on the L&F. When organized in a structure called a ButtonGroup, only one ButtonGroup in the group can be selected at one time. JRadioButtons are Mutually Exclusive.

Just as with the JCheckBoxes, JRadioButtons have many Constructors, taking an Icon, a String caption, and a preset boolean for either Selected or Deselected. When a JRadioButton is Selected, it fires an ItemEvent. Therefore, if you use a JRadioButton, you must implement the ItemListener interface as with the JCheckBox. The logical relationship between the buttons gathered together to form a button group is maintained by the object ButtonGroup.

Warning: The ButtonGroup is NOT a GUI object. It cannot appear on the GUI and so you do NOT ever add a ButtonGroup object to the Container. To do so is a syntax error. Joining your JRadioButtons into a ButtonGroup is entirely separate from adding the JRadioButtons to the GUI's Container.

```
Container c = getContentPane();
c.setLayout( new FlowLayout );
JRadioButton ital = new JRadioButton( "Italic" );
JRadioButton bold = new JRadioButton( "Bold" );
JRadioButton undr = new JRadioButton( "Underline" );
c.add( ital );
c.add( bold );
```

```
c.add( undr );
ButtonGroup bg = new ButtonGroup();
bg.add( ital );
bg.add( bold );
bg.add( undr );
```

The process of collecting JRadioButtons into a ButtonGroup is entirely separate from the creation of the GUI.

10.9 JMenuBar, JMenu, JMenuItem

The menu bar is an aspect of modern GUIs that users have become very common. The tool bar is just as common. Creating and managing either one requires significant effort. There are two broad categories of menus:

- Menu Bars (on the top) (Known as a top-level menu) (Menus that are contained in another menu are called pull-right menus.)
- Popup Menus (coming from anywhere else)

Menus can only be attached to objects of classes that contain the method: `setJMenuBar()`

If you understand the terms used to describe a menu, building one can be very easy.

The bar across the top is the JMenuBar. After you instantiate the single JMenuBar, you set it to be the JMenuBar. To place a JMenu object in the JMenuBar, you add it. A JMenu item always has children. If this object is the end of the road, it is a JMenuItem . Because this has children menus, it is a JMenu .

Example:

```
JMenuBar bar = new JMenuBar();
setJMenuBar( bar );
action = new JMenu( "Action" );
enterNew = new JMenuItem( "Enter New Data" );
enterNew.addActionListener( this );
printData = new JMenu( "Print Data" );
printDataSorted = new JMenuItem( ”Printed in Sorted
Order" );
printData.add( printDataSorted );
printDataEntry = new JMenuItem( ”Print in Entry Order"
);
printData.add( printDataEntry );
exit = new JMenuItem( "Exit" );
```

```
action.add( enterNew );
action.add( printData );
action.add( exit );
bar.add( action );
```

Because menus and menu bars (also toolbars) are all subclasses of the JComponent class, they are all lightweight containers. Since they're containers, any type of component can be added to a Swing menu such as the icons you saw in the example.

10.10 JPopupMenu

A Popup Menu comes from anywhere on the GUI. On a Windows L&F, the 2nd mouse button triggers it. On a Unix Mofif L&F, the 3rd mouse button triggers it. The ActionListener has to work hard to determine if the correct mouse button was pressed—meaning, if it was a valid "popup trigger".

10.11 JToolBar

Unlike the menus, a JToolBar is not added with a specific method. Rather, the JToolBar is just added to the container like any other component. Further, Tool bars should be added to a container that contains an instance of the BorderLayout. In this BorderLayout, the JToolBar should be put in anywhere but the center of the BorderLayout.

To begin, the JToolBar is a container that can be oriented either horizontally or vertically. A Swing Tool Bar can float—meaning it can move around anywhere on the GUI the user desires. Using JToolBar methods:

- add(Action), and
- addSeparator,

any type of component can be added to a JToolBar

This is a sample JToolBar. We just treat it as a component that gets buttons added to it. We can add any component to it, although the results don't always look the greatest unless we tweak the layout. If you drag a JToolBar, you can cause it to float into another side, where it will "stick" and attach itself. Otherwise, it can pop up in its own window.

10.12 JList

This class presents a GUI list of selectable objects. It supports three selection modes:

- single selection: you choose only one thing at a time.
- single interval selection: you choose several things in a row.

• multiple interval selection: you choose several clumps of items that are not continuous. This is the default setting.

The JList class delegates the job of showing the list and handling its events. Therefore, the list's model keeps a list of objects, that are rendered in list "cells" by the (VIEW), a "list cell renderer."

A JList can display an Icon and a String—but not both at the same time. The DefaultListCellRenderer can only display a single object at one time. If you implement your own custom subclass from the JList , you can make it do special things. Right out of the box, the JList does not support scrolling, but—just as with the JTextArea—you can add it to a JScrollPane object. You place the JList inside a JScrollPane , and the list will scroll. When the user clicks on an item in the JList, a ListSelectionEvent occurs.

```
Object[] items = { "item1", "item2", "item3", "item4" );
JList list = new JList( items );
list.setVisibleRowCount( 4 );
list.setSelectionMode(    ListSelectionModel.SINGLE_
SELECTION );
list.addListSelectionListener( this );
public void valueChanged( ListSelectionEvent e )
{
}
```

Instantiate your JList. Set parameters, register the ListSelectionListener as your listener and override the inherited abstract method from the Listener interface, and use the information contained in the ListSelectionEvent object. (Only one method to override)

Any complexity that arises in this object comes from the chance that multiple rows may have been selected.

Constructor:

```
ListSelectionEvent(       Object source,
int firstIndex,
int lastIndex,
boolean isAdjusting
);
```

In practice, you would use the same getSource() that you have used for the ActionEvent object. Both ActionEvent and ListSelectionEvent inherit this same getSource() method from EventObject. However, you will need to cast this back into a JList object to really take advantage of the information it contains.

If you wish to see what values in the list were selected when the ListSelectionEvent was triggered, you use either of two methods:

```
public Object getSelectedValue()
public Object[] getSelectedValues()
```

In a fully rigorous usage, this is the way a JList is implemented.

```
public void valueChanged( ListSelectionEvent e )
{
if( e.getValueIsAdjusting() )
return;
JList listEvent = (JList) e.getSource();
if( listEvent.isSelectionEmpty() )
{
// This means nothing was selected.
else
{
int index = listEvent.getSelectedIndex();
// This returns the location of the single line
// that was selected from the list.
}
}
```

If the user is still choosing, this allows us to wait. We can cast the event from a generic object into a specific one that we can probe.

```
public void valueChanged( ListSelectionEvent e )
{
String s = "";
if( e.getValueIsAdjusting() )
{
s = "Value is adjusting";
}
else
{
s = "Selection from e.getFirstIndex() + "to" +
e.getLastIndex();
}
showStatus( s );
}
```

For this example, we don't have to cast it into a JList object. We can use the

methods in the ListSelectionEvent object. When you move through a list, this is the place you where you started. When you're making a selection, this is the place where you stopped.

With a JList, you have many alternative ways to discover what the user did:

JList Selection Properties

Property - Meaning

- anchor —the index that most recently began an interval selection.
- lead —the last index of the most recent selection interval.
- maximum —the highest index from the selected items.
- minimum —the lowest index from the selected items.

The JList component delegates three major jobs to other objects:

- data handling - to ListModel
- item selection - to ListSelectionModel
- cell rendering - to ListCellRenderer

All three of these are interfaces.

Example:

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
public class Test extends JApplet
{
public void init()
{
Container c = getContentPane();
c.setLayout(new FlowLayout());
Object[] items = { "item one", "item two", "item three",
"item four", "item five", "item six", "item seven", "item
eight", "item nine", "item ten" };
JList list = new JList( items );
list.setVisibleRowCount( 7 );
c.add( list );
}
}
```

Instantiate an array of any Object. Then you instantiate the JList object with your array. You usually decide how many rows of the list you want to see at any one time. The default visible size is 8 rows.

Remember, when we just add the JList reference to the Container, we do not automatically get scrolling. For that, you need to add the JList to a JScrollPane. Notice, although we specified a visible limit of 7 rows—we're seeing 10. What gives? The setVisibleRowCount() method only takes effect when the list is in a scroll pane.

You recall, the JList delegates its labor to other classes that actually do the work. For example, the JList does not manage the data handling—the references—to the objects it displays. Instead, all instances of JList delegate the management of the data to an object that implements the ListModel interface. To facilitate this model, you can construct instances of a JList using the following Constructors:

- public JList(ListModel)
- public JList(Object[])
- public JList(Vector)

After you have constructed an instance of the class JList, you can change what is displayed with the following:

- public void setModel(ListModel)
- public void setListData(Object[])
- public void setListData(Vector)

These methods allow you to change the entire contents of the list in one motion. Changing individual elements is not so easy. Since the JList does not maintain its own data, it cannot help you change individual elements in the list. To accomplish that, you have to go to where that data is stored.

- You have to directly manipulate the list's model.
- All of these "models" must implement the ListModel interface.

"This interface defines the methods that components like JList use to get the value of each cell in a list and the length of the list." Logically the model is a vector. Its indexes vary from 0 to ListDataModel.getSize() - 1. Any change to the contents or length of the data model must be reported to all of the

10.12.1 ListDataListeners—the controllers

Adding or Removing An Item from the JList

Whenever you add or remove an item from your list, that is another type of event. To track that event, you use the object

ListDataListener

- If you want to change an item in a JList, you have to get its model, and go through the interface

ListDataListener

Methods the implementing classes must override:

- public abstract void addListDataListener(ListDataListener)
- public abstract void removeListDataListener(ListDataListener)

Classes that implement this interface are notified when the contents of the list changes.

One of the three methods defined by the ListDataListener interface class is invoked whenever data associated with a list is modified. Methods the implementing classes must override:

- public abstract void contentsChanged(ListDataEvent)
- public abstract void intervalAdded(ListDataEvent)
- public abstract void intervalRemoved(ListDataEvent)

If just one item is changed, the first method is called. When an "interval" or several contiguous objects in the list are added, either of the last two methods are called.

This event object has three constants that define the change that occurred.

- public static final int CONTENTS_CHANGED
- public static final int INTERVAL_ADDED
- public static final int INTERVAL_REMOVED

Constructors:

public ListDataEvent(Object source, int type, int index(), int index1)

This method will tell you the number of rows or items in the list.

public abstract int getSize()

This method will give you a reference to a specific Object in the list.

public abstract Object getElementAt()

10.13 JComboBox

A JComboBox gets its name because it is a combination of an editable area and a drop-down list of selectable items. In other words, a JComboBox is a more complex JList. Because both display a list of items, they both have models that extend the ListModel interface. Likewise, both are components that that render list cells by implementing the ListCellRenderer interface.

Instances of the JComboBox are—by default—not editable, so you just have to make a simple call to:

```
JComboBox.setEditable( true )
```

and you can edit to your heart's content.

This example is not editable, as you can see by the gray background in the JComboBox. The behavior of these two variants—editable and not editable—is significantly different.

Code Necessary for a JComboBox :

```
private JComboBox comboBox = new JComboBox();

comboBox.addItem( "Top" );
comboBox.addItem( "Center" );
comboBox.addItem( "Bottom" );
comboBox.setEditable( true );
comboBox.getEditor().addActionListener(       new
ActionListener()
{
public void actionPerformed(ActionEvent e)
{
System.out.println("here"  +  comboBox.getSelectedItem()
);
}
}
);
```

This example uses a default constructor. You could also have used:

- public JComboBox(ComboBoxModel)
- public JComboBox(Object[])
- public JComboBox(Vector)

The most common way you will use a JComboBox is for making another object react to the choices expressed by the user. For this purpose, you implement the

ItemListener interface.

```
String[] numbers = { 'one', 'two', 'three', 'four' };
JComboBox cb = new JComboBox( numbers );
cb.setMaximumRowCount( 3 );
cb.addItemListener( new Itemlistener()
{
public void itemStateChanged( ItemEvent e )
{
someJLabel.setText( numbers[ cb.getSelectedIndex() ] );
}
}
);
cb.getSelectedIndex()
```

This method returns an int, which becomes a subscript for the array,

which returns a String, which becomes the text.

Event handling in a JComboBox is tricky for several reasons. When their selected item changes—either by being selected, or by having them be edited— every JComboBox fires two events.

1.For deselecting the previous item.

2.For selecting the current item.

Also, whenever an item event is fired, an action event is also fired right away.

Because Combo Boxes can be edited, they can fire ActionEvents that you can track with an ActionListener. In fact, whenever the currently displayed value is edited, the combo box fires an action event.

```
public class Test extends JApplet
{
private JComboBox comboBox = new JComboBox();
private ComboBoxEditor editor = comboBox.getEditor();
public void init()
{
Container c = getContentPane();
comboBox.setEditable(true);
comboBox.addItem("Top");
comboBox.addItem("Center");
comboBox.addItem("Bottom");
c.setLayout(new FlowLayout());
c.add(comboBox);
editor.addActionListener( new ActionListener()
{
public void actionPerformed(ActionEvent e)
{
String s = (String) editor.getItem();
showStatus("Item Edited: " + s);
}
});
}
}
```

This is an interface, it has method getItem(), which returns an Object. Once we get an Object out of the editor, we cast it into type String, and discover what the edit was that triggered the event. As you might imagine, this is a highly customizable component.

10.14 JTextArea

This familiar component can display multiple lines. It is basic in its function. For complex abilities, choose the JEditorPane and JTextPane. It also does not implement scrolling by itself. You can specify its word wrapping behavior.

```
JTextArea txt = new JTextArea();
txt.setLineWrap( true );
```

Properties maintained by the JTextArea:

- columns: int—the number of columns displayed
- line count: int—the number of lines of text contained in the text area.
- line wrap: boolean—determines whether or not lines of text are wrapped at the right edge of a text area.
- rows: int—the number of lines of text displayed in a text area.
- tabSize: int—number of characters inserted when the tab key is pressed.
- wrapStyleWord: boolean—when true, causes words to be wrapped on words. If false, wrap is on characters.

10.15 JEditorPane

Just like JTextAreas, these are capable of displaying multiple lines of editable text. Unlike text areas, JEditorPanes can display HTML and RTF formats.

```
JEditorPane editor = new JEditorPane();
editor.setPage( url );
```

Hyperlink events are fired when a mouse pressed event occurs over a hyperlink in an editor pane that is displaying an HTML document. Naturally, Swing defines a HyperLinkListener interface and HyperLinkEvent class that permits hyperlink events to be handled.

```
public abstract void
hyperlinkUpdate( HyperlinkEvent h )
```

The HyperlinkEvent object offers the following methods:

HyperlinkEvent:

- public String getDescription()
- public HyperlinkEvent.EventType getEventType()
- public URL getURL()

Example:

```
JEditorPane editor = new JEditorPane();
...
editor.addHyperlinkListener(
new HyperlinkListener()
```

```
{
public void hyperlinkUpadate( HyperLinkEvent e )
{
editor.setPane( e.getURL() );
}
}
);
```